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 – 2017

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um *Visualizador Científico* simples. Ao final do projeto será possível gerar gráficos a partir de dados obtidos por um microcontrolador.

O objetivo do EP1 é exercitar os conceitos da Orientação a Objetos ao implementar um software para visualizar *textualmente* os dados capturados por um microcontrolador.

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 FRDM-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.

Este projeto será desenvolvido incrementalmente e em dupla nos três Exercícios Programas de PCS 3111. Para este primeiro EP o *Visualizador Científico* deve apresentar textualmente os dados de séries obtidas pelo microcontrolador, seguindo a especificação deste documento. A interface com o microcontrolador já está implementada, devendo ser apenas usada.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo e método – o que representa o conteúdo até a Aula 3. 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 **Ponto** e **Serie** definidas a seguir, além de criar um **main** que permita o funcionamento do programa como desejado. O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. Não é permitido criar outras classes ou definir outros métodos para as classes **Ponto** e **Serie** além dos especificados a seguir. **Mas você deve definir os atributos dessas classes, conforme necessário**.

Não será avaliado neste EP a distinção de visibilidade pública e privada (assunto da Aula 4); todos os métodos e atributos devem ser públicos.

2.1 Classe Ponto

Um **Ponto** é um par de valores. Um dos valores é a coordenada horizontal (x) e o outro é a coordenada vertical (y). Deve ser possível imprimir um **Ponto** para facilitar sua visualização. Também deve ser possível dizer se dois **Pontos** são iguais.

A classe **Ponto** deve possuir os seguintes métodos (não deve possuir outros métodos):

```
/**
 * Obtêm o valor do Ponto na coordenada horizontal (x).
 */
double getX();

/**
 * Obtêm o valor do Ponto na coordenada vertical (y).
 */
double getY();

/**
 * Define o valor do Ponto na coordenada horizontal (x).
 */
void setX(double x);

/**
 * Define o valor do Ponto na coordenada vertical (y).
 */
void setY(double y);

/**
 * Imprime na saída padrão (cout) o ponto no formato (x, y).
 * Pule uma linha após imprimir cada Ponto.
 */
void imprimir();

/**
 * Informa se este ponto é igual a outro.
 * Um ponto é igual se os valores x e y dos Pontos são
 * suficientemente próximos.
 */
bool eIgual(Ponto* outro);
```

Os métodos `getX` e `getY` devem retornar os valores das coordenadas horizontal e vertical do ponto, respectivamente; os métodos `setX` e `setY` devem definir esses valores.

A respeito do método `eIgual`, um ponto é suficientemente próximo a outro[1] se a diferença entre os valores das coordenadas x dos dois pontos é menor que um ϵ e a diferença entre os valores das coordenadas y dos dois pontos é menor que um ϵ. Considere $\epsilon = 10^{-5}$ (`1e-5` em C++)[2].

No caso do método `imprimir`, para o **Ponto** (2, 3) a saída deve ser:

```
(2, 3)
```

Note que deve haver um pula linha após imprimir o **Ponto**. Ou seja, a saída ao imprimir em seguida os Pontos (2, 3) e (4, 5) deve ser:

```
(2, 3)
(4, 5)
```

[1] Isso é necessário devido à representação de um número ponto flutuante em um computador.

[2] Para mais detalhes sobre esse problema veja https://isocpp.org/wiki/faq/newbie#floating-point-arith.

2.2 Classe Serie

Uma **Serie** é uma sequência de **Pontos** cujos valores são obtidos por canais (fonte de dados) informados pelo microcontrolador. Um canal deve prover os valores da coordenada horizontal (x) da **Serie** e um outro canal deve prover os valores da coordenada vertical (y) da **Serie**. A **Serie** deve armazenar quais canais foram usados como fonte de dados (isso será útil futuramente). Deve ser possível adicionar novos **Pontos** à **Serie** e obter os **Pontos** anteriormente adicionados.

A classe **Serie** deve possuir os seguintes métodos (não deve possuir outros métodos):

```
#define NUMERO_MAXIMO_VALORES 10

// Permite definir o nome, o nomeDoCanalX e o nomeDoCanalY.
void setNome(string nome);
void setNomeDoCanalX(string nomeDoCanalX);
void setNomeDoCanalY(string nomeDoCanalY);

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

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

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

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

/**
 * Obtém 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.
 *
 * Caso a Serie não tenha valores, deve-se retornar NULL.
 */
Ponto* getLimiteSuperior();

/**
 * Obtêm 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.
 *
 * Caso a serie nao tenha valores, deve-se retornar NULL.
 */
Ponto* getLimiteInferior();

/**
 * Obtêm o ponto que está na posição definida da Serie. A contagem de
 * posições começa em 0.
 *
 * Em caso de posições inválidas, retorne NULL.
 */
Ponto* getPosicao(int posicao);

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

Os métodos `getNome`, `getNomeDoCanalX` e `getNomeDoCanalY` obtêm os valores do nome, do nome do canal X e do nome do canal Y, respectivamente. Os métodos "set" respectivos definem esses valores.

O método `adicionar` deve adicionar valores até o `NUMERO_MAXIMO_VALORES`, uma constante já definida em "Serie.h". 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: eles só não devem ser adicionados).

O método `getPosicao` deve obter o **Ponto** que está na posição informada. A contagem de posições deve começar em `0` e, em caso de posições inválidas, o método deve retornar NULL. O método `getQuantidade` deve informar a quantidade de **Pontos** que foram adicionados à **Serie** e o método `estaVazia` deve informar se a **Serie** está vazia ou não.

O método `getLimiteSuperior` deve retornar um objeto **Ponto** que tem como coordenada x o maior valor horizontal existente na **Serie** e o maior valor vertical existente na **Serie** como coordenada y. Caso a **Serie** não possua valores, esse método deve retornar NULL. De forma similar, o método `getLimiteInferior` deve retornar um **Ponto** com os menores valores horizontal e vertical da **Serie**, ou NULL caso a **Serie** não possua valores.

Para exemplificar o funcionamento dos métodos `getLimiteSuperior`, `getLimiteInferior` e `getPosicao`, considere a Serie `{(2, 3), (5, 1), (1, 2)}`. Nesse caso o método `getLimiteSuperior` deve retornar o Ponto `(5, 3)`, o método `getLimiteInferior` deve retornar o Ponto `(1, 1)` e `getPosicao(0)` deve retornar `(2, 3)` e `getPosicao(2)` deve retornar `(1, 2)`.

O método `imprimir` deve imprimir a Serie no seguinte formato:

```
Serie <nome da série>
Ponto 1
Ponto 2
...
```

Por exemplo, para a Serie com nome "ACCX" e valores `{(2, 3), (5, 1), (1, 2)}`, o resultado da impressão deve ser (note o '\n' no final):

```
Serie ACCX
(2, 3)
(5, 1)
(1, 2)

```

3 Arquivos Entregues e Projeto do CodeBlocks

Para facilitar o desenvolvimento e a correção, as classes foram separadas em arquivos diferentes. Como esse é um assunto da Aula 4, está disponível no e-Disciplinas um projeto do CodeBlocks com os arquivos necessários e as diretivas necessárias em cada um deles.

No arquivo "Ponto.h" e "Serie.h" você deve apenas adicionar os atributos necessários às respectivas classes, ou seja, eles já possuem as definições dos métodos. Nos arquivos "Ponto.cpp" e "Serie.cpp" você deve implementar os métodos especificados. No arquivo "main.cpp" você deve implementar a função `main`, considerando a interface com o usuário explicada na Seção 4.

Além desses arquivos é também entregue a classe **InterfaceSerial** ("InterfaceSerial.h" e "InterfaceSerial.cpp"). Essa classe permite obter dados do microcontrolador. O uso dessa classe e a interface com o microcontrolador é explicado na Seção 5.

4 Interface com o usuário

Por simplicidade, a interface com o usuário neste EP será através do console. **A ordem das mensagens informadas pelo usuário deve seguir exatamente a ordem definida**, assim como o formato das respostas esperadas.

O `main` deve pedir para o usuário o nome da **Serie** e permitir que ele escolha o canal para as coordenadas X e o canal para as coordenadas Y, através da lista de canais informadas pela **InterfaceSerial**. Em seguida o main deve perguntar o número de pontos que o usuário deseja obter. Com isso, crie um objeto **Serie** e adicione nele os valores obtidos pelos canais através da **InterfaceSerial**. Depois de obter os pontos, imprima a **Serie** e os pontos de limite superior e inferior.

Um exemplo de entrada e saída é apresentado abaixo, indicando em vermelho os valores informados pelo usuário. Note que ACCX, ACCY, ACCZ, MAGX, MAGY e MAGZ foram informados pela placa e, portanto, variam dependendo do programa que roda na placa.

```
Aperte o botao reset da placa.
Informe o nome da serie: Aceleracao
Escolha o canal X:
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
1
Escolha o canal Y:
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
2
Obter quantos pontos? 5
Obtendo os pontos
Imprimindo os pontos obtidos
Serie Aceleracao
(0.0002, -0.0684)
(0, -0.0688)
(-0.0012, -0.0691)
(-0.0012, -0.0691)
(-0.0005, -0.0684)

Limite Superior: (0.0002, -0.0684)

Limite Inferior: (-0.0012, -0.0691)
```

Note que o usuário deve escolher a lista de canais a partir do 1 (e não do 0).

Com isso, o `main` deve possuir o seguinte estilo:

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

// Obtém o nome da Serie e os canais escolhidos
...

// Obtém o número de Pontos a adicionar
int quantidade;
...
```

```
// Cria a Serie
...

cout << "Obtendo os pontos" << endl;
for (int i = 0; i < quantidade; i++) {
  is->atualizar();
  // adiciona os valores à Serie
  ...
}

cout << "Imprimindo os pontos obtidos" << endl;
...
cout << "Limite Superior: ";
...
cout << "Limite Inferior: ";
...
```

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

5 Interface com o microcontrolador

O programa permite apresentar valores obtidos de um microcontrolador, o disponível no Kit Freescale Freedom FRDM-KL25Z. Os dados serão obtidos usando a porta serial, conectada por um cabo USB. Para isso é necessário um programa na placa e a classe **InterfaceSerial** 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 (configurado como 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. Os métodos que você pode usar dessa classe são os apresentados a seguir:

```
/**
 * Inicializa a interface serial.
 */
void inicializar(string porta);

/**
 * Obtém o valor lido para o canal na última atualização.
 */
double getValor (string canal);
/**
 * Espera por ESPERA ms e atualiza os valores dos canais.
 *
 * Em caso de problema de leitura, retorna false. Caso
 * contrário, retorna true.
 */
bool atualizar();

/**
 * Obtêm um arranjo com todos os nomes dos canais disponíveis
 */
string* getNomeDosCanais();

/**
 * Informa a quantidade de canais disponíveis.
 */
int getQuantidadeDeCanais();
```

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.

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 retorna uma sequência de valores para dois canais ("T1" e "T2"). No "InterfaceSerial.h" deixou-se apenas:

[3] Existem soluções mais elegantes.

```
#ifndef INTERFACESERIAL_H
#define INTERFACESERIAL_H

#include <string>

using namespace std;
class InterfaceSerial {
public:
  void inicializar(string porta);
  double getValor (string canal);
  bool atualizar();
  string* getNomeDosCanais();
  int getQuantidadeDeCanais();
private:
  int numeroDeChamadas = -1;
  string nomes[2];
};
#endif
```

No "InterfaceSerial.cpp" foi usada a seguinte implementação:

```
#include "InterfaceSerial.h"

void InterfaceSerial::inicializar(string porta) {
  nomes[0] = "T1";
  nomes[1] = "T2";
}

double InterfaceSerial::getValor(string canal) {
  int valoresT1[] = {10, 20, 30, 40};
  int valoresT2[] = {10, 15, 20, 25};

  if (canal == "T1") return valoresT1[numeroDeChamadas];

  return valoresT2[numeroDeChamadas];
}

bool InterfaceSerial::atualizar() {
  numeroDeChamadas++;
  return true;
}

string* InterfaceSerial::getNomeDosCanais() {
  return nomes;
}

int InterfaceSerial::getQuantidadeDeCanais() {
  return 2;
}
```

Note que a implementação faz *diversas* simplificações. Por exemplo, a classe informa apenas 4 valores para os canais. Cabe ao testador criar a classe que teste o que ele quer da forma mais simples possível e, claro, não usar a classe fora dos limites que ele mesmo definiu!

6 Entrega

O projeto deverá ser entregue até dia **13/09** em um Judge específico, disponível em <http://judge.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 e-Disciplinas 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). Entregue todos os arquivos, inclusive o main (que deve obrigatoriamente ficar em um arquivo "main.cpp"), em um arquivo comprimido. Os fontes não devem ser colocados em pastas. O Judge fará uma verificação *básica* do software, verificando se é possível instanciar as classes **Ponto** e **Serie**.

Não altere o nome dos arquivos entregues: "Ponto.h", "Ponto.cpp", "Serie.h", "Serie.cpp", "main.cpp", "InterfaceSerial.h" e "InterfaceSerial.cpp".

7 Dicas

- Não adicione, de forma nenhuma e por motivo nenhum, métodos públicos às classes. Você pode (e deve) adicionar atributos.
- A classe **InterfaceSerial** deve ser entregue da mesma forma que vocês a receberam. Isso não impede que você faça alterações para testes – é só não entregar com essas alterações.
- A entrada de dados para o main deve seguir exatamente a ordem apresentada.
- Teste com diversos valores diferentes. Considere também valores que podem causar erros.
- Implemente a solução aos poucos – não deixe para implementar tudo no final.