ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E SISTEMAS DIGITAIS
PCS

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

Exemplo de Prova

Instruções

- A prova é **INDIVIDUAL**. Não consulte outras pessoas.
- Qualquer forma de plágio será punida com nota 0.
- Uma vez iniciada a prova, haverá 140 minutos para fazer as submissões no Judge. Submissões após este período não serão consideradas.
- Deverão ser feitas **CINCO** submissões no Judge; cada submissão trata de um problema que corresponde a um item do enunciado. Para todas as submissões, envie todos os arquivos ".h" e ".cpp", inclusive o "main", em um arquivo ".zip".
- A submissão de cada exercício pode ser feita até *5 vezes* sem penalização. Para cada submissão subsequente, a nota máxima do exercício será decrementada em 2 pontos: 6ª. Submissão, nota máxima 8,0; 7ª. Submissão, nota máxima 6,0. ...

Critério de avaliação

- Correção automática: 6,0
- Aplicação dos conceitos de Orientação a Objetos e atendimento à especificação: 3,0
- Qualidade do código (indentação, adequação dos nomes, etc.): 1,0

Os espaços de *coworking* permitem que diversas empresas compartilhem um mesmo escritório. O proprietário do espaço divide as salas e mesas de trabalho, cobrando um valor de cada empresa dependendo de quanto espaço ela está usando.

Deseja-se criar um software para calcular o preço de aluguel de um espaço de *coworking*. Para isso foram criadas as classes `Mesa`, `Sala`, `SalaDeReuniao`, `SalaDeTrabalho`, `SalaDeTrabalhoCompartilhada`.

Junto com o enunciado há um projeto do Code::Blocks **já configurado** contendo um *esqueleto* destas classes, notadamente os seus métodos públicos. Em especial, a classe `Mesa` já está pronta. **Não altere-a**.

Deve-se complementar a definição dos atributos das classes e depois implementá-las. **Não crie classes além dessas**. O enunciado define os métodos públicos que as classes devem **obrigatoriamente** possuir. Adicione os atributos necessários, compatíveis com a especificação, sempre seguindo o conceito de *encapsulamento*. Se forem necessários, crie também métodos auxiliares que não sejam públicos. **As classes não devem possuir outros membros (atributos ou métodos) públicos além dos especificados**. No caso das classes `SalaDeReuniao`, `SalaDeTrabalho` e `SalaDeTrabalhoCompartilhada`, pode-se adicionar métodos da superclasse para redefini-los. Todas as exceções definidas são da biblioteca padrão.

Junto com as classes foram entregues dois arquivos: `testeP2.cpp` e `main.cpp`. O arquivo `testeP2.cpp` contém a função de teste que será usada para verificar a saída (item "e") – assim como em aula. O arquivo `main.cpp` tem a função `main`, que invoca a função `testeP2`. Pode-se submeter o arquivo `main.cpp` sem problemas – o Judge ignorará esse arquivo. Use a função `testeP2` para testar as suas classes, devendo ser implementada conforme definido no código.

CUIDADO

- Submeta **TODOS** os arquivos "`.h`" e "`.cpp`" nos itens (inclusive o `testeP2.cpp`). A correção manual do código será feita no último item submetido.

Dicas

- Compile e teste com frequência!
- Faça o `main` aos poucos, para testar cada um dos itens implementados.
- Pode-se implementar cada item mesmo se as classes solicitadas nos itens anteriores não estiverem funcionando. O Judge usa um gabarito dessas classes para efeito de avaliação.

(a) **[1,0 ponto]** Implemente a classe `Sala`

A classe **abstrata** `Sala` representa uma sala que pode ser utilizada pelas empresas no *coworking*. A definição dos métodos públicos da classe `Sala` é apresentada a seguir, sendo omitidos os *defines* e *includes*.

```
class Sala {
public:
    Sala(string nome, int metragem);
        virtual ~Sala();

        string getNome();
        int getMetragem();
    void imprimir();
        double getPreco();
};
```

Complete a definição, colocando atributos se necessário, e a respectiva implementação. O construtor da classe recebe como parâmetro um string `nome`, que identifica a sala (por exemplo "C1-38") e um inteiro que define a metragem (dimensão em metros quadrados, por exemplo 77m^2). Caso o parâmetro `metragem` seja menor ou igual a 0 deve ser jogada uma exceção `invalid_argument` (da biblioteca padrão) com a mensagem "Metragem invalida".

São definidos também os métodos *getters* para os atributos definidos no construtor.

O método `imprimir` já está implementado. Descomente-o ao implementar a classe.

O método getPreco deve ser **abstrato**. Para testar essa classe, você pode implementar o item "b".

(b) **[1,0 ponto]** Implemente a classe `SalaDeReuniao`.

A `SalaDeReuniao` é uma classe concreta filha de `Sala` em que podem ser realizadas reuniões. O preço de aluguel da sala é definido pela sua dimensão, tendo métodos com `escopo de classe` para definir tais valores.

A seguir são apresentados os métodos públicos e específicos desta classe:

```
class SalaDeReuniao {
public:
  SalaDeReuniao(string nome, int metragem);
  virtual ~SalaDeReuniao();

  static void setPrecoPorMetroQuadrado(double valor);
  static double getPrecoPorMetroQuadrado();
};
```

O construtor recebe o nome e a metragem, assim como na classe mãe `Sala`.

O preço da sala deve ser calculado ao multiplicar o preço por metro quadrado com a metragem da sala. Por exemplo, caso o preço por metro quadrado seja R$ 10,0 e a sala tenha 15 m^2, o método `getPreco` deve retornar R$ 150,0.

O preço por metro quadrado das salas de reunião é definido usando um atributo estático. **Defina-o com valor inicial de R$ 10,0**. O valor do preço por metro quadrado deve ser obtido pelo método estático `getPrecoPorMetroQuadrado`; o valor deve ser definido pelo método `setPrecoPorMetroQuadrado`. No caso do método `setter`, caso o valor seja menor que 0 deve ser jogada uma exceção `invalid_argument` com a mensagem "Valor invalido".

(c) **[1,5 ponto]** Implemente a classe `SalaDeTrabalho`.

A `SalaDeTrabalho` é uma classe concreta filha de `Sala` que possuem mesas (classe `Mesa`). O preço de aluguel da sala é definido pela quantidade de mesas disponíveis. **Não defina** atributos como `protected` (faça todos privados, para não impactar o teste do item "d").

```
class SalaDeTrabalho {
public:
    SalaDeTrabalho(string nome, int metragem);
        virtual ~SalaDeTrabalho();

        void adicionar(Mesa* m);
        vector<Mesa*>* getMesas();
};
```

Essa classe possui os mesmos parâmetros no construtor da sua classe mãe: o nome e a metragem. No destrutor, destrua todas as mesas adicionadas e o `vector` criado.

O método `adicionar` deve permitir adicionar a mesa à `SalaDeTrabalho`. Porém, o método não deve permitir adicionar uma mesa já adicionada anteriormente (compare

com "==" os objetos), jogando uma exceção `logic_error` com a mensagem "Mesa ja existente". Também não é possível adicionar uma nova mesa caso não haja mais espaço na sala (caso ao tentar adicionar a mesa a soma das metragens das mesas com a metragem da nova mesa seja maior do que a metragem da sala), jogando também uma exceção `logic_error` com a mensagem "Mesa nao cabe".

As mesas adicionadas à sala devem ser retornadas pelo método `getMesas`, o qual retorna um `vector`. Portanto, use um `vector` para armazenar as mesas.

O preço desse tipo de sala deve ser calculado pelo número de mesas multiplicado por 100. Portanto, para 5 mesas, o aluguel da sala deve custar R$500,00.

(d) **[1,5 ponto]** Implemente a classe `SalaDeTrabalhoCompartilhada`.

A classe `SalaDeTrabalhoCompartilhada` é uma classe concreta filha de `SalaDeTrabalho`. Ela permite compartilhar mesas com outras empresas e, portanto, o seu preço depende de quantas mesas são usadas.

A definição dos métodos públicos desta classe são apresentados a seguir.

```
class SalaDeTrabalhoCompartilhada {
public:
    SalaDeTrabalhoCompartilhada(string nome, int metragem);
        virtual ~SalaDeTrabalhoCompartilhada();

        void reservar(Mesa* m);
        void reservar();

        vector<Mesa*>* getMesasReservadas();
};
```

Assim como na `SalaDeTrabalho`, o construtor recebe o nome e a metragem da sala. No destrutor não se esqueça de destruir o `vector` criado (mas cuidado: não destrua duas vezes as mesas!).

O método `reservar` deve reservar uma `Mesa` dessa sala. Caso a mesa já esteja reservada, o método deve jogar um `logic_error` com a mensagem "Mesa ja reservada". Caso a mesa não faça parte da sala (ou seja, ela não tenha sido adicionada à sala), o método deve jogar uma exceção do tipo `invalid_argument` com a mensagem "Mesa nao faz parte da sala". Dica: utilize a função `find` (do cabeçalho `algorithm`) para detectar se uma mesa faz parte da sala.

O cálculo de preço deve considerar o número de mesas reservadas, sendo o número de mesas *reservadas* multiplicado por 85. Por exemplo, em uma sala com 10 mesas, mas com apenas 2 reservadas, o preço deve ser R$ 170,00.

O método `reservar` sobrecarregado, sem parâmetro, deve reservar a sala toda. Para isso, faça um laço varrendo todas as mesas da sala invocando o método `reservar(Mesa*)`. Trate ainda no próprio método, através de um comando try-catch, as possíveis exceções do tipo `logic_error` de mesas já reservadas jogadas por `reservar(Mesa*)`.

O método `getMesasReservadas` deve retornar um `vector` com todas as mesas já reservadas.

(e) **[1,0 ponto]** No arquivo `testeP2.cpp`, implemente uma função `testeP2` que irá testar as suas classes.

A função `testeP2` deve executar os seguintes passos:

(1) Crie uma sala de reunião com nome "A1" e 100m^2, colocando-a em uma variável do tipo `Sala`, e a imprima.

(2) Altere o valor do preço por metro quadrado para 15 das salas de reunião. Imprima novamente a sala de reunião com nome "A1".

(3) Crie uma sala de trabalho com nome "A2" e 30m^2, colocando-a numa variável do tipo `Sala`. Adicione as seguintes mesas à essa sala:

Mesa com id 1 e tamanho de 10m^2

Mesa com id 2 e tamanho de 12m^2

Imprima a sala de trabalho com nome A2.

(4) Crie uma sala de trabalho compartilhada com nome "A3" e 30m^2, colocando-a em uma variável do tipo `SalaDeTrabalhoCompartilhada`. Adicione as seguintes mesas:

Mesa com id 3 e tamanho de 10m^2

Mesa com id 4 e tamanho de 12m^2

Reserve somente a mesa com id 4 e imprima a sala de trabalho com nome A3.

(5) Destrua todas as salas (a ordem da destruição não é relevante)

O resultado esperado é apresentado abaixo:

```
Sala A1 (100) - R$ 1000
Sala A1 (100) - R$ 1500
Sala A2 (100) - R$ 200
Sala A3 (30) - R$ 85
Mesa 1 destruida
Mesa 2 destruida
Mesa 3 destruida
Mesa 4 destruida
```

TESTES DO JUDGE

Item A

- Classe Sala eh abstrata
- Construtor metodos get e imprimir
- Construtor metragem menor ou igual a zero
- Metodo getPreco abstrato

Item B

- SalaDeReuniao eh filha de sala
- Construtor e metodos get
- Metodos setPrecoPorMetroQuadrado e getPrecoPorMetroQuadrado sao estaticos
- Valor padrao do preco por metro quadrado
- Metodos get e set do PrecoPorMetroQuadrado
- setPrecoPorMetroQuadrado com valor menor que 0
- Metodo getPreco
- Metodo getPreco alterando preco por metro quadrado depois de construir

Item C

- SalaDeTrabalho eh filha de Sala
- Construtor e metodos get
- Adicionar 1 mesa e getMesas
- Adicionar varias mesas e getMesas
- Adicionar mesa repetida no comeco
- Adicionar mesa repetida no meio
- Adicionar mesa repetida no fim
- Adicionar mesa limite de espaco
- Adicionar mesa sem espaco
- Adicionar mesa sem espaco no limite
- Destrutor destroi mesas
- getPreco com varias mesas
- getPreco intercalado com adicionar

Item D

- SalaDeTrabalhoCompartilhada eh filha de SalaDeTrabalho
- Construtor Destrutor e metodos get
- Reservar 1 mesa
- Reservar varias mesas
- Reservar sala toda sem reservas anteriores
- Reservar sala toda com reservas anteriores
- Reservar 1 mesa ja reservada
- Reservar varias mesas ja reservadas
- Reservar mesa que nao faz parte da sala
- getPreco com varias mesas reservadas
- getPreco intercalado com reservar

Item E

- testeP2 item 1
- testeP2 item 2
- testeP2 item 3
- testeP2 item 4
- testeP2 item 5