Universidade Federal de São Carlos

Trabalho Integrado

Programação Orientada a Objetos e Estrutura de Dados I

Gestão de Cinema

Autores:	*Registros Acadêmicos:*
Alessandro V. Piccoli	380105
Douglas H. dos Santos	552640
Jonathan A. N. da Silva	489557
Lucas L. Silva	552321
Lucas S. Rocha	552330

Profa. Dra. Katti Faceli
Profa. Dra. Tiemi Christine Sakata

Zaina Software

Sorocaba, 30 de Junho de 2014

Sumário

1 Introdução

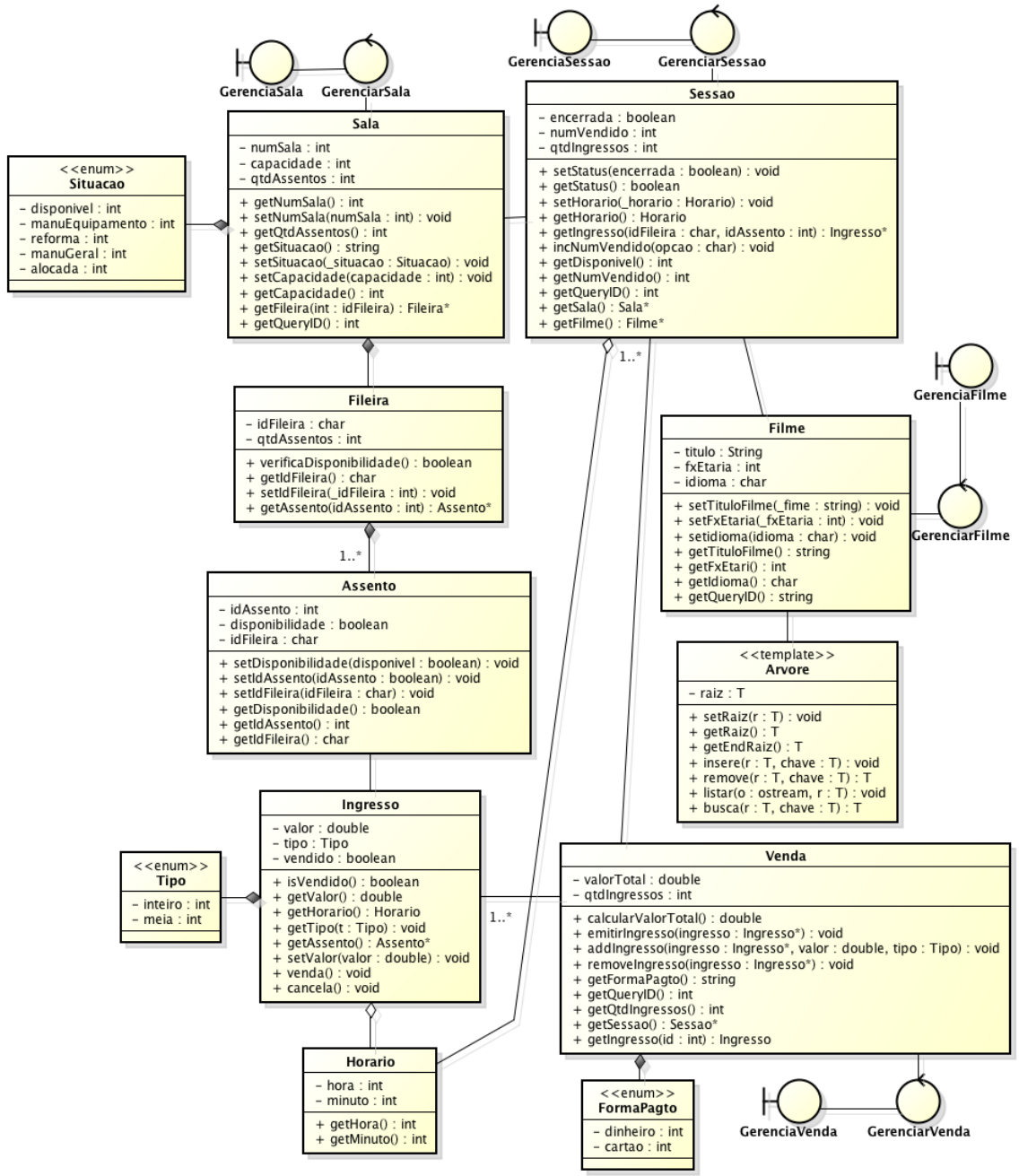
O programa foi desenvolvido com a finalidade de dar suporte à gestão de um cinema para a empresa **Faceli & Sakata Ltda**, esta gerencia uma rede de salas de cinemas. O cenário foi definido de forma que o sistema será desenvolvido por uma segunda empresa: **Zaina Software**, que é consolidada na área de desenvolvimento de *software*, e já fez produtos para outras redes de cinema no estado de São Paulo.

O programa apresenta como principais funcionalidades: o gerenciamento das salas do cinema, filmes, sessões e vendas de ingressos. Uma rede de cinema pode ter muitas salas de exibição, sendo necessário registrar e manter as informações de cada uma delas (capacidade, número de fileiras e de assentos por fileira junto com a numeração):

- Cada sala pode ter um número de fileiras e de assentos por fileira diferentes e pode apresentar um ou mais filmes em diferentes sessões.
- Uma sessão é caracterizada por um filme, uma sala e um horário e tem um número máximo de ingressos que podem ser vendidos de acordo com a capacidade da sala.
- Um ingresso deve conter informações sobre seu tipo: meia entrada ou inteira.
- Um cliente só pode comprar ingressos de sessões ainda não encerradas.
- Uma venda pode ser referente a mais de um ingresso e o comprador pode solicitar vários assentos um ao lado do outro.
- O comprador indica quantos assentos ele quer, e o sistema deve mostrar todas as opções com esse número de assentos consecutivos.
- Quando há algum problema em uma das salas este deve ser registrado.
- Além do gerenciamento, o *software* deve possibilitar a visualização de relatórios.

2 Descrição e Diagrama de classes

Considerando o cenário e os requisitos funcionais aprensentados, foi elaborado o diagrama de classes abaixo:



A nossa Equipe foi contratada para fazer a implementação do sistema projetado. Para isso, as seguintes regras foram seguidas:

- Todo o tratamento de erros deve ser feito pelo mecanismo de tratamento de exceção;
- O código deve estar bem documentado, e deve ser feita também uma documentação externa;
- Acrescentar os atributos e métodos que forem necessários;
- Ao iniciar e finalizar a execução, o sistema deve recuperar e armazenar em arquivo todos os dados já cadastrados. Empregar os operadores <<e >>;
- Para controlar os objetos das classes, é necessário definir e implementar as estruturas de dados de nossa escolha.

3 Como Utilizar o Programa

3.1 Menu e Métodos de Entrada

O nosso programa foi organizado por menus, onde você escolhe a opção de operação desejada, e o índice encontrado à esquerda da operação representa o número que deverá ser inserido no programa. Qualquer outro número que esteja fora dos especificados no menu não serão aceitos como válidos, e a tela com a lista de operações será reexibida.

Ao iniciar o programa você encontrará o menu principal, onde poderá estar escolhendo um tipo de gerência dentre as existentes no projeto, através desse menu você poderá realizar desde cadastros de assentos até buscas de filmes, para isso, deverá escolher a sessão correspondente ao tipo de operação que você deseja.

Após escolher uma das sessões correspondentes, você entrará em um novo menu, onde poderá estar realizando todos os tipos de operações referentes ao CRUD da sessão, podendo ser este desde cadastro de uma nova venda, busca de salas, listagem de sessões ou remoção de filmes. As quatro operações estarão disponíveis a todos so tipos de gerenciamento.

3.2 Tipos de Operações:

- Criar: você poderá estar criando um novo objeto ou serviço através dessa operação. É importante ter conhecimento de que, para muitos serviços, é necessário haver ao menos 1 cadastro de algum objeto, um exemplo é no caso de pesquisar uma sessão que não existe, ou efetuar uma venda sem filmes cadastrados. Dependendo do tipo de cadastro a entrada pode ser diferente, variando entre caracteres ou algarismos, por isso é importante estar atento ao que é requisitado na hora do cadastro, e ser inserido os dados corretos, para que não haja problemas de busca ou remoção futuramente.
- Remover: como o nome já diz, esta operação remove algum dado ou objeto cadastrado no sistema. Uma observação importante é que, assim como você não possui nenhum cadastro no início do programa, não podendo

assim efetuar operações como busca. Caso você remova por completo algum objeto (remover todos os filmes por exemplo), você também entrará na situação descrita, onde determinados serviços estarão indisponíveis.

- Buscar: você poderá sempre imprimir na tela a lista de operações ou servidos cadastrados ou calculdaos. Mesmo os dados sendo salvos nos arquivos ".data", é de crucial importância que seteja disponível a exibição das informações durante a execução do programa, para isso essa operação foi desenvolvida e estará sempre ativa (mas nem sempre disponível, como descrito anteriormente é necessário que algumas condições sejam efetivadas para o funcionamento correto).

- Editar: após algum tipo de cadastro, pode ser que informações tenham sido inseridas incorretamente e seja necessário corrigi-las, para que não o usuário não precise remover e cadastrar novamente os dados, criamos um menu para edição de informações no sistema. Após a edição ser finalizda, os dados são atualizados tanto no software (enquanto este está sendo executado) como nos arquivos ".data", onde os novos valores serão salvos.

- Listar: um fato importante em qualquer projeto, é como e quando poderão ser exibidos relatórios para o usuário. No nosso projeto é possível exibir uma lista de todos os objetos e serviços cadastrados, uma forma de relatório, onde é o usuário acompanha o avanço do funcionamento e cadastros no programa. Cada gerenciador terá a sua listagem separada, exibindo os dados pertinentes ao mesmo. Como em outros casos ja descritos, apenas serão exibidos relatórios de informações ja cadastradas, caso não exista nenhum serviço ou objeto cadastrado, o comando de listagem não será executado e um alerta será exibido.

4 Implementações

O projeto foi desenvolvido em linguagem C++, utilizando o paradigma orientado a objetos, possibilitando assim a modelagem do projeto em classes as quais permitem uma visão do sistema semelhante à uma visão do mundo real de um cinema. Utilizaremos estas classes para realizar os procedimentos necessários na venda de ingresssos e gerenciamento do cinema em geral.

A seguir listaremos as classes utilizadas no projeto, uma descrição simples sobre cada uma delas (papel no sistema e comportamento) e por fim o algoritmo da classe criada pelo Grupo (*header*).

4.1 Classe Sessão

A Sessão é uma das classes mais importantes do projeto, responsável, de certo modo, pelo gerenciamento geral do projeto. Está conectada à todos os "braços"do diagrama de classes, se relacionando, direta ou indiretamente, com a maioria das classes envolvidas no projeto.

Podemos dizer que é a estrutura base para o projeto, possuindo maior parte da informação necessária (através dos ponteiros que permitem que esta classe conheça muitas outras) para realização das vendas e controle dos dados.

Como é possível ver, utilizamos métodos para manipulação das informações e relacionamentos entre as classes que estão de alguma forma relacionadas à esta classe.

4.1.1 Algoritmo Desenvolvido

```
#ifndef SESSAO_H
#define SESSAO_H

#include <iostream>
#include "../headers/consts.h"
#include "../headers/sala.h"
#include "../headers/horario.h"
#include "../headers/filme.h"
#include "../headers/ingresso.h"

using std::string;

class Sessao
{
  private:
  int id;
    Horario horario;
    bool encerrada;
    int numVendido;
    Filme *filme; // uma sessao conhece um filme
    Sala *sala; // uma sessao conhece uma sala
    Ingresso **ingressos; // uma sessao possui ingressos
        relativos a ela
    int qtdIngressos;

  public:
```

```
    Sessao(Sala *_sala, int _id, Horario _horario, Filme *
        _filme);
    ~Sessao();

    void setStatus(bool encerrada);
    bool getStatus();

    void setHorario(Horario _horario);
    Horario getHorario();

    Ingresso* getIngresso(char idFileira, int idAssento);

    void incNumVendido(char opcao);
    int getDisponivel();
    int getNumVendido();

    int getQueryID();

    Sala* getSala();
    Filme* getFilme();
};

#endif // SESSAO_H
```

4.2 Classe Sala

A classe Sala gerencia as informações referentes a cada sala dentro do cinema. É composta pela classe Fileira (relação 1 para muitos), criando também um vetor de ponteiros para Fileira ao ser criada. Possui os dados referentes à situação e capacidade das salas. Uma sala também tém conhecimento da disponibilidade de cada assento que pertence a cada fileira que a compõe.

4.2.1 Algoritmo Desenvolvido

```
#ifndef SALA_H
#define SALA_H

#include <iostream>
#include "../headers/fileira.h"

using std::string;

enum Situacao {disponivel, manuEquipamento, reforma,
    manuGeral, alocada};

class Sala
{
  private:
    int numSala;
    int capacidade;
    int qtdAssentos;
    Situacao situacao;
    Fileira **fileiras; // vetor de ponteiros para
        fileiras

  public:
    Sala(int _numSala, int _capacidade, Situacao _situacao
        , int _qtdAssentos);
    ~Sala();

    int getNumSala();
    void setNumSala(int numSala);

    int getQtdAssentos();
    string getSituacao();
    void setSituacao(Situacao _situacao);

    void setCapacidade(int capacidade);
    int getCapacidade();

    Fileira* getFileira(int idFileira);

    int getQueryID();
};

#endif // SALA_H
```

4.3 Classe Fileira

A classe Fileira é a responsável pela criação dos assentos que formam uma sala.

Com essa classe podemos recolher todas as informações sobre cada assento através do método getAssento(), para depois enviar estas informações para a classe Sala.

4.3.1 Algoritmo Desenvolvido

```
#ifndef FILEIRA_H
#define FILEIRA_H

#include <iostream>
#include "../headers/assento.h"

class Fileira
{
  private:
    char idFileira;
    int qtdAssentos;
    Assento **assentos; // vetor de ponteiros para
        assentos

  public:
    Fileira(char _idFileira, int _qtdAssentos);
    ~Fileira();
    bool verificaDisponibilidade();
    char getIdFileira();
    void setIdFileira(int _idFileira);
    Assento* getAssento(int idAssento);
};

#endif // FILEIRA_H
```

4.4 Classe Assento

Classe capaz de instanciar todos os assentos de cada fileira da sala. Quando um ingresso é vendido, é esta classe que altera a situação do assento referente à este ingresso através dos métodos de gerenciamento da disponibilidade. A partir da disponibilidade de cada assento podemos analisar a disponibilidade da fileira, e a partir desta ver o ocupamento da sala.

4.4.1 Algoritmo Desenvolvido

```
#ifndef ASSENTO_H
#define ASSENTO_H

#include <iostream>

class Assento
{
  private:
    int idAssento; // num do assento na fileira
    bool disponibilidade; // 1 se disponivel, 0 caso
        contrario
    char idFileira; // num da fileira da qual este assento
         faz parte

  public:
    Assento(int _idAssento, bool _disponibilidade, char
        _idFileira);
    void setDisponibilidade(bool _disponibilidade);
    void setIdAssento(int _idAssento);
    void setIdFileira(char _idFileira);
    bool getDisponibilidade(); // eh disparado pela classe
         fileira
    int getIdAssento();
    char getIdFileira();
};

#endif // ASSENTO_H
```

4.5 Classe Filme

A classe Filme é responsável pelo gerenciamento de cada filme cadastrado. Estes objetos possuem todas as informações necessárias para a descrição de um filme, tais como título, faixa etária, idioma, etc.

Para instanciar o conjunto de filmes existentes em nosso cinema, foi utilizado um *template* da estrutura de dados Árvore Binária Balanceada (ABB) desenvolvido pelo nosso grupo. A justificativa para o uso desta estrutura de dados é o fato desta estrutura permitir buscas em tempo otimizado, além de ser capaz de manter os filmes em ordem alfabética por natureza, sem necessidade de utilizar algoritmos de ordenação, o que é muito interessante em termos de complexidade.

4.5.1 Algoritmo Desenvolvido

```
#ifndef FILME_H
#define FILME_H
#include <iostream>
#include <fstream>
using std::string;
using std::ostream;
using std::ofstream;

class Filme
{
  friend ostream & operator<<(ostream &o, const Filme &f);
  friend ofstream & operator<<(ofstream &o, Filme &f);
  private:
    string tituloFilme;
    int fxetaria;
    char idioma;  // L - legendado; D -  dublado; N -
        nacional

  public:
    Filme(string _tituloFilme = " ", int _fxetaria = 0,
        char _idioma = ' ');
    Filme(const Filme &f);
    void setTituloFilme(string _filme);
    void setFxEtaria(int _fxEtaria);
    void setIdioma(char idioma);
    string getTituloFilme();
    int getFxEtaria();
    char getIdioma();
    string getQueryID();
    // sobrecargas necessarias para funcionamento da
        arvore
    bool operator<(Filme &f);
    bool operator>(Filme &f);
    bool operator==(Filme &f);
    bool operator!=(Filme &f);
};
#endif // FILME_H
```

4.6 Classe Ingresso

Esta é outra classe com grande importância dentro do projeto, por estar ligada à diversas outras classes. Cada objeto desta classe possui toda informação necessária sobre os ingressos que são vendidos e emitidos no cinema.

Como pode ser visto no Diagrama de Classes, esta classe se relaciona diretamente com a Venda (pois cada venda realizada é de um ou mais ingressos, que possuem as informações de um assento para o cliente, com isso, podemos ver claramente a ligação com a classe Assento), mas além desta, é necessário um relacionamento com a classe Sessão, pois, segundo nosso *design* de projeto, quando uma sessão é instanciada, automaticamente geramos os ingressos relativos à todos assentos da sala que esta sessão aloca, e a disponibilidade destes é atualizada sempre que realizamos uma venda

Os métodos venda() e cancela() desta classe se relacionam diretamente com o assento referente a cada ingresso. Ao chamar venda(), mudamos o status deste assento para ocupado, e ao chamar cancela(), fazemos este assento voltar a ser disponível. Estes métodos são disparados sempre que a classe Venda solicita uma adição ou remoção de ingresso.

4.6.1 Algoritmo Desenvolvido

```
#ifndef INGRESSO_H
#define INGRESSO_H
#include <iostream>
#include "../headers/horario.h"
#include "../headers/assento.h"
using std::string;

enum Tipo {inteiro, meia};

class Ingresso{
  private:
    Horario horaIngresso;
    double valor;
    Tipo tipo;
    Assento *assento;
    bool vendido;

  public:
    Ingresso(Horario _horaIngresso, Assento *_assento);

    bool isVendido(); // retorna se o ingresso foi vendido
        ou nao
    double getValor();
    Horario getHorario();
    string getTipo();
    void setTipo(Tipo t);
    Assento* getAssento();
    void setValor(double valor);
    void venda(); // ocupa o assento
    void cancela(); // libera o assento
};
#endif // INGRESSO_H
```

4.7 Classe Venda

A responsabilidade principal de um objeto da classe Venda é obter as informações referentes à uma única venda (que pode conter vários ingressos), enviar informações para o controle das sessões (disponibilidade de assentos), e enviar mensagens para todos ingressos relativos à esta venda contendo informações como tipo do ingresso e valor.

4.7.1 Algoritmo Desenvolvido

```
#ifndef VENDA_H
#define VENDA_H

#include <iostream>
#include <iomanip>
#include "../headers/consts.h"
#include "../headers/ingresso.h"
#include "../headers/sessao.h"

using std::string;
using std::setprecision;
using std::fixed;

enum FormaPagto{dinheiro, cartao};

class Venda{

  private:
  int id;
    double valorTotal;
    FormaPagto formaPagto;
    Ingresso **ingressos; // vetor de ponteiros para
        ingressos
    int qtdIngressos;
    Sessao *sessao; // atualiza disponibilidade

  public:
    Venda(Sessao *_sessao, int _id ,FormaPagto _formaPagto
        );
    ~Venda();
    double calcularValorTotal();
    void emitirIngresso(Ingresso *ingresso);
    void addIngresso(Ingresso *ingresso, double valor,
        Tipo tipo);
    void removeIngresso(Ingresso *ingresso);
    string getFormaPagto();

    int getQueryID();
    int getQtdIngressos();
    Sessao *getSessao();
    Ingresso *getIngresso(int id);
};

#endif // VENDA_H
```

4.8 Classe Horario

Esta classe foi criada com o intuito de implementar uma definição simples e robusta de um horário. Ela conta com apenas dois atributos, hora e minuto, que definem um horário, e também conta com sobrecargas feitas especialmente para ler e imprimir horários no formato HH:MM, que é muito intuitivo para qualquer usuário do sistema.

4.8.1 Algoritmo Desenvolvido

```
#ifndef HORARIO_H
#define HORARIO_H

#include <iostream>
#include <sstream>

using std::ostream;
using std::istream;
using std::stringstream;

class Horario
{
  friend ostream &operator<<(ostream &o, const Horario &h)
      ;
  friend istream &operator>>(istream &i, Horario &h);
  friend stringstream &operator>>(stringstream &s, Horario
       &h);

  private:
    int hora;
    int minuto;

  public:
    int getHora();
    int getMinuto();
    Horario() : hora(0), minuto(0) {} ;
    Horario(int h, int m) : hora(h), minuto(m) {} ;
    Horario(const Horario &h)
    {
      hora = h.hora;
      minuto = h.minuto;
    }
};

#endif // HORARIO_H
```

4.9 Boundary GerenciaSessao

Boundary responsável por gerenciar a classe Sessao, seja para criar ou remover uma sessão ou salvar as informações cadastradas no arquivo ".data", no qual estaremos realizando o *backup* das informações fornecidas de forma organizada, nos perditindo trablahar de maneira mais segura e eficiente.

Como pode ser observado pelo algoritmo a seguir, os métodos compostos nessa boundary são de grande maioria referentes ao CRUD da classe Sessao, assim como os atributos são ponteiros usados para gerenciar a memória que será usada para salvar as informações da classe (referentes à sala e filme).

4.9.1 Algoritmo Desenvolvido

```
// BOUNDARY - fronteira
#ifndef GERENCIASESSAO_H
#define GERENCIASESSAO_H

#include <iostream>
#include "../headers/sessao.h"
#include "../headers/consts.h"
#include "../headers/gerenciasala.h"
#include "../headers/gerenciafilme.h"

class GerenciaSessao{
  private:
    int qtdSessoes;
    Sessao *sessoes[MAX_SESSOES];
    GerenciaSala *gerencSala;
    GerenciaFilme *gerencFilme;

  public:
    GerenciaSessao(GerenciaSala *g, GerenciaFilme *f);
    /* ------ CRUD ------- */
    void criarSessao(); // retorna true se criou com
        sucesso
    void removerSessao(); // retorna true se removeu com
        sucesso
    void buscarSessao();
    Sessao* buscarSessao(int id);
    void editarSessao();
    /* ------------------- */
    void escreverSessao(); // funcao para escrever no
        arquivo sessoes.data as salas cadastradas
};

#endif // GERENCIASESSAO_H
```

4.10 Boundary GerenciaSala

Boundary responsável por gerenciar a classe Sala, tendo o mesmo propósito que o boundary descrito anterimente, este é usado para o CRUD da sala, criando, removendo, buscando ou editando informações da classe.

Os atributos usados nesse boundary por sua vez, não se restringem apenas à ponteiros, onde iremos armazenar as informações de cada sala, mas também uma variável com a quantidade de salas, utilizada para maior administração da classe.

4.10.1 Algoritmo Desenvolvido

```
// BOUNDARY - fronteira
#ifndef GERENCIASALA_H
#define GERENCIASALA_H

#include <iostream>
#include "../headers/sala.h"
#include "../headers/consts.h"

class GerenciaSala{
  private:
    int qtdSalas;
    Sala *salas[MAX_SALAS];

  public:
    GerenciaSala();
    /* ------ CRUD ------- */
    void criarSala(); // retorna true se criou com sucesso
    void removerSala(); // retorna true se removeu com
        sucesso
    void buscarSala();
    Sala* buscarSala(int _id);
    void editarSala();
    /* ------------------- */
    void escreverSala(); // funcao para escrever no
        arquivo salas.data as salas cadastradas
};

#endif // GERENCIASALA_H
```

4.11 Boundary GerenciaVenda

A classe de vendas, sendo uma das que recebe maior movimento de informações, também possui um boundary, necessário para que cada venda aconteça sem problemas.

Quando se realiza uma venda é necessário estarmos atentos a vários fatores importantes, que variam dos assentos ao ingresso em si, por isso a classe é composta por métodos para adicionar ingressos à venda, e nos atributos possuimos o espaço preparado na meḿoria para o armazenamento das informações, uma variável para controle da quantidade de vendas realizadas e um ponteiro para o boundary da Sessão, pois este possui informações sobre todo o cinema, muitas vezes essencial para que a venda ocorra.

4.11.1 Algoritmo Desenvolvido

```
// BOUNDARY - fronteira
#ifndef GERENCIAVENDA_H
#define GERENCIAVENDA_H

#include <iostream>
#include "../headers/consts.h"
#include "../headers/ingresso.h"
#include "../headers/sessao.h"
#include "../headers/venda.h"
#include "../headers/gerenciasala.h"
#include "../headers/gerenciasessao.h"

class GerenciaVenda
{
  private:
    Venda *vendas[MAX_VENDAS];
    int qtdVendas; // por sessao e por horario
    GerenciaSala *gerencSala;
    GerenciaSessao *gerencSessao;
    void escreverVenda();

  public:
    GerenciaVenda(GerenciaSala *gerencSala, GerenciaSessao
        *gerencSessao);
    void criaVenda();
    void addIngressoVenda(int idVenda);
    void removeIngressoVenda(int idVenda);
    void exibirVenda(int idVenda);
};

#endif // GERENCIAVENDA_H
```

4.12 Boundary GerenciaFilme

O grupo concordou que seria de grande utilizade o desenvolvimento de um boundary para a classe Filme, pelo simples fato de que cada filme possui um CRUD, e este deve ser organizado e administrado de forma clara e prática, possibilitando o cadastro ou exclusão de filmes com métodos específicos, além é claro, de atributos que proporcionem o armazenamento e a organização das informações na memória de forma segura.

É importante ressaltar que os dados estão sendo armazenados em arquivos ".data", onde as informações podem ser consultadas e salvas de maneira prática.

4.12.1 Algoritmo Desenvolvido

```
// BOUNDARY - fronteira
#ifndef GERENCIAFILME_H
#define GERENCIAFILME_H

#include <iostream>
#include "../headers/consts.h"
#include "../headers/filme.h"
#include "../headers/arvore.h"

class GerenciaFilme{
  private:
    int qtdFilmes;
    Arvore<Filme> filmes;
    void escreverFilme(); // funcao para escrever no
        arquivo Filmes.data os Filmes cadastradas

  public:
    GerenciaFilme();
    /* ------ CRUD ------- */
    void criarFilme(); // retorna true se criou com
        sucesso
    void removerFilme(); // retorna true se removeu com
        sucesso
    void buscarFilme();
    Filme* buscarFilme(string _tituloFilme);
    void editarFilme();
    /* ------------------- */
    void listarFilmes();
};

#endif // GERENCIAFILME_H
```

5 Estrutura de Dados Usada

Durante o projeto, utilizamos alguns métodos aprendidos em sala para gerenciamento dos dados, principalmente quanto à parte de busca, onde utilizamos árvores binárias de busca (ABB) para manter os filmes do cinema e permitir que as operações sobre estes filmes sejam eficientes (buscas em ABB possuem eficiência da ordem de O($log\ n$), e quanto ao armazenamento, onde discutimos os métodos mais fáceis e eficazes nos casos que trabalhamos, tanto para informações inseridas pelo usuário como dados gerenciados durante a execução do projeto.

O arquivo **Arvore.h** representa um *template* de ABB com métodos de inserção, remoção, busca e listagem *inorder*, lembrando que este último realiza a impressão ordenada devido à própria natureza de uma ABB. No caso dos filmes, a ordem é dada pela ordem lexicográfica dos títulos.

Foi considerado o uso da estrutura de árvore AVL devido ao auto-balanceamento deste tipo de árvore, que permite uma eficiência ainda maior nas buscas. Porém, por dificuldade de implementação, e devido ao fato de o ganho de eficiência não ser muito grande, foi mantida a utilização das árvores ABB, cuja implementação é bem mais simples.

6 Como Compilar

Para facilitar a compilação, utilizamos um Makefile que reconhece todos os arquivos de implementação necessários (.cpp) e os compila automaticamente. Basta usar o comando make no terminal. Ao finalizar a compilação, será gerado um executável nomeado "cinema.exe". Para desinstalá-lo, basta usar o comando make uninstall.
O funcionamento do Makefile é explicado a seguir.

1. Variáveis
   Nós criamos duas variáveis antes de executar as targets no algoritmo. Primeiramente utilizamos a "CC", que estará recebendo como parâmetro uma referência indicando o tipo de compilador que será necessário para a criação do arquivo executável do nosso projeto, sendo neste caso, o compilador "g++", pois estamos compilando um algoritmo desenvolvido em C++. Em segundo lugar utilizamos de uma variável "EXE", que recebe uma string para o nome do arquivo de output da compilação do projeto.

2. Target referenciando arquivo de output
   Nessa fase do Makefile, nós estamos referenciando nossa variável que contém o arquivo de output e a target "clean", para limpeza do diretório.

3. Execução da Compilação
   Após os procedimentos descritos anteriormente, estamos prontos para realizar enfim a compilação. Primeiramente nós chamamos a variável "CC"com o tipo de compilador a ser usado e declaramos a pasta onde estão todos os arquivos ".cpp". Segundo nós realizamos o mesmo tipo de operação com a classe "main", cujo pré-requisito é a necessidade de ja possuir todas as outras classes ja compiladas e prontas para uso. Quando executamos essa parte do Makefile é gerado o arquivo executável, tornando nosso projeto pronto para ser usado.

4. Target Clean
   Dentro dessa target nós especificamos os comandos necessários para limpar um arquivo, removendo todos os arquivos objeto (*.o) das bibliotecas, além de arquivos temporários ( ). É importante lembrar que após a execução da compilação essa target é executada automaticamente.

5. Target Uninstall
   Essa target é apenas um acréscimo ao nosso Makefile, tornando possível a desinstalação do arquivo executável. Sua chamada não é implícita, portanto para ser utilizado basta escrever "uninstall"após a chamada do Makefile.

6.1 Makefile

```
CC = g++
EXE = cinema.exe

all: $(EXE) clean

$(EXE):
  $(CC) classes/*.cpp -c
  $(CC) main.cpp *.o -o $(EXE)

clean:
  rm -r -f *.o
  rm -r -f *~

uninstall:
  rm -f $(EXE)
```

7 Decisões de Projeto

Durante a implementação do projeto, nosso Grupo realizou diversas decisões relacionadas à complementações ou modificações do mesmo. Estas decisões foram realizadas tanto em partes da estrutura geral do projeto (diagrama de classes) como em classes específicas (interface e responsabilidades de cada classe).

Com estas decisões pudemos criar um projeto mais detalhado e organizado, com operações mais controladas e eficazes.

Iremos descrever a seguir cada uma das decisões, seguidas pela especificação de qual parte do projeto elas foram implementadas e o porque de cada decisão tomada.

7.1 Referentes à Classe Sessão

1. Atributo Encerrada
   Realizamos a troca do tipo do atributo de Inteiro para Booleano, pois haverão apenas dois casos possíveis (encerrada; não encerrada).

2. Ponteiro tipo Sessão
   Ao invés de possuir um ponteiro de ponteiro, criamos apenas um ponteiro para um vetor, que terá cada campo ligado à um assento diferente.

3. Acrescentamos uma Struct para horários
   Optamos por criar uma struct composta por variáveis específicas para horas e minutos ao invés do método nativo de usar a biblioteca time. Esta decisão foi tomada pelo fato de que a biblioteca possuia muitos itens que não seriam utilizados, foi mais viável a criação da struct.

4. Utilização do Heapsort
   Pelo fato de utilizarmos um vetor, a implementação com o Heapsort se torna mais fácil, além de a complexidade do mesmo ser O($n \log n$).

5. Criação de um vetor de horários
   Decidimos por utilizar um vetor com 48 posições para armazenar todos os horários da sessão. O tamanho do vetor se dá para um tratamento de exceção (1 filme a cada 30 minutos por 2 dias seguido).

7.2 Referentes à Classe Sala

1. Switch case para exemplificar a interface
   No case formSalas, teremos a Sala Padrão, Sala 3D e Criar Personalizada. Em cada caso terá uma variável que será passada para o construtor respectivo.

2. Atributo qtdFileiras
   Especificação Futura

3. Atributo qtdAssentos
   Todas as fileiras terão necessariamente a mesma quantidade de assentos sempre.

4. Método getFileira
   Utilizamos o método para retornar o vetor de Fileira.

5. Capacidade da Sala
   Variável utilizada para definir a quantidade de fileiras na sala, delimitando assim a capacidade total da sala.

7.3 Referentes à Classe Fileira

1. Observação quanto ao Construtor
   Construtor irá definir quantidade de assentos nas fileiras.

2. Atributo qtdAssentos
   Decidimos fazer a criação de um novo atributo responsável pela quantidade de assentos em uma fileira.

3. Método getAssento
   Utilizamos o método para retornar um determinado assento.

7.4 Referentes à Classe Assento

1. Atributo Disponibilidade
   Realizamos a troca do tipo do atributo de Inteiro para Booleano, pois haverá apenas dois casos possíveis (disponível; não disponível).

2. Complemento dos atributos idAssento, idFileira e Disponibilidade
   Acrescentamos os métodos get e set para todos os métodos.

7.5 Referentes à Classe Venda

1. Função emitirIngresso
   Através da função (tendo como parâmetro "ingresso:Arrays"e retorno Void) iremos imprimir na tela as informações referentes aos ingressos adicionados (método addIngressos com ingresso:Ingresso como parametro). Vale lembrar que pode ser adicionado mais de um ingresso.

7.6 Referentes à Classe Main

1. Atributo qtdHorarios
   Utilizamos desse atributo para controlar a quantidade de horários de uma sessão.

2. Atributo maxHorarios
   Decidimos criar esse atributo para ter o controle da quantidade máxima de horários que a sessão pode ter.

3. Atributo maxSessoes
   Através deste atributo controlaremos o tamanho máximo de número de sessões que podem haver.

7.7 Referentes à Classe Filme

1. Criação da Classe
   Observando o diagrama de classes, decidimos que seria uma boa ideia para o desenvolvimento do projeto que fosse criado uma classe separada "Filmes", onde estariam localizadas todas as informações sobre cada filme cadastrado.

8 Dificuldades no Projeto

Durante o desenvolvimento do projeto, devido ao pouco tempo de implementação efetiva com a linguagem C++, tivemos algumas dificuldades relativas à sintaxe e uma implementação adequada que mantesse o encapsulamento de todas as classes, acessando seus atributos e mantendo suas alterações como esperado.

Além disso, houve dificuldade em relação à escolha das estruturas de dados que seriam potencialmente necessárias para nosso projeto. Foi cogitada a utilização de listas ligadas e de árvores AVL, porém, por simplicidade, fizemos uso apenas de árvores ABB. De qualquer forma, vale citar que o uso de árvores AVL para manter a busca de salas, sessões e filmes seria viável e interessante.

9 Testes Realizados

Para testar todas as funcionalidades do sistema, simulamos uma execução em que adicionamos uma sala, uma sessão, alguns filmes (sendo que um deses é relativo à sessão criada), e uma venda de alguns ingressos para esta sessão. Mais detalhes sobre os testes realizados, e novos testes presenciais serão realizados durante a apresentação do projeto.

10 Conclusão

Este trabalho integrado nos proporcionou uma experiência de se trabalhar em grupo, dividindo as responsabilidades, assim como num ambiente empresarial, que enfrentaremos no mercado de trabalho futuramente. Para isto utilizamos um repositório privado no servidor do GitHub, onde todos os membros poderiam ter acesso e onde conseguiríamos um controle de versões.

Durante esta forma de trabalhar, que é nova para alguns de nossos integrantes de projeto, pudemos perceber o quanto os conceitos de programação orientada a objetos são comuns no dia-a-dia e são de extrema importância para a modelagem de problemas do mundo real em sistemas de informação, reúso de código, separação de responsabilidades e divisão da estrutura de um projeto.

Conseguimos aplicar os conhecimentso passados em sala de aula de forma efetiva e em uma situação real, aplicando, assim, todos os conceitos de POO dentro de um sistema que lida com a realidade, mostrando a sua importância e implementação.

Tais fatores auxiliaram no aprendizado dos conceitos básicos aprendidos durante o semestre e permitiram com que qdquiríssemos maior entendimento da linguagem e conhecimento de forma geral.

Outro ponto muito positivo foi o estímulo de se trabalhar com um grupo unido e com um mesmo foco, e quando o objetivo é cumprido, dividir este sucesso de um projeto funcional é extremamente gratificante.

11 Bibliografia

Referências

[1] AGUILAR, LUÍS J. – Programação em C ++: algoritmos, estruturas de dados e objetos – Porto Alegre: McGraw Hill, 2008.

[2] HARVEY M. – C++ como programar – 5ed. São Paulo: Pearson, 2006.

[3] FACELI, K. Slides de aula – Programação Orientada a Objetos, 2014.

[4] HORSTMANN, C. – Conceitos de computação com o essencial de C++ – 3ed. Porto Alegre: Bookman, 2005.

[5] SUTTER H. – Programação avançada em C++ – 1ed. São Paulo: Pearson, 2006.