**Relatório do Problema 2: “Sistema para cadastro de espectadores do filme *Star Wars*”**

**Valmir Vinícius de Almeida Santos**

Engenharia de Computação – Universidade Estadual de Feira de Santana  
Feira de Santana, 07 de março de 2016

vvalmeida96@gmail.com

**1. Introdução**

Um dos grandes acontecimentos no cenário do cinema global em 2015 foi o retorno da aclamada franquia *Star Wars*. O lançamento, cujo título em português é: “Star Wars: O Despertar da Força”, conquistou o recorde de terceira maior bilheteria da história, além de ser a maior bilheteria da franquia. Diversos cinemas ao redor do mundo precisaram se preparar para atender a grande demanda de fãs em busca de ingressos para as sessões de exibição do filme.

Esse relatório objetiva retratar e detalhar o processo de modelagem e desenvolvimento de um sistema a ser utilizado na unificação da venda de ingressos para a obra cinematográfica em todo o Brasil. A solução implementada é responsável por controlar o processo de comercialização de ingressos e gerenciar os dados das entidades diretamente envolvidas nessa atividade, como compradores e cinemas (com as suas respectivas salas e sessões). Além disso, o software é capaz de fornecer listagens dos registros e organizar a distribuição de camisas temáticas para os compradores registrados no fã clube oficial da franquia.

A aplicação foi desenvolvida com base na programação orientada a objetos, utilizando a linguagem Java. Buscando-se a materialização de um código fonte mais organizado, o padrão MVC *(Model-View-Controller)* foi adotado. Além disso, estruturas de dados foram empregadas para o armazenamento de informações em tempo de execução. A principal foi a lista encadeada. Outra estrutura implementada na solução foi a fila, entretanto, essa foi utilizada mais como um mecanismo de organização do que de armazenamento propriamente dito. O mecanismo aplicado para ordenação desses dados foi o algoritmo *MergeSort*.

**2. Fundamentação teórica**

Nessa sessão serão apresentadas os elementos teóricos que foram utilizados como base durante a modelagem e a implementação do projeto.

**2.1. Programação orientada a objetos**

Programação orientada a objetos é um método de implementação no qual programas são organizados como coleções colaborativas de objetos. Cada um dos objetos representa uma instância de determinada classe, as quais fazem parte de uma hierarquia organizada por relacionamentos de herança (BOOCH et al., 2007).

**2.1.1. Herança**

Um dos principais recursos do paradigma de programação orientada a objetos é a herança. Deitel (2005, p. 301) diz que: ” A herança é uma forma de reutilização de software na qual uma nova classe é criada, absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas. Com a herança, os programadores economizam tempo durante o desenvolvimento do programa reutilizando software de alta qualidade, testado e depurado.”

Com a utilização da herança, ao criar uma classe o programador designa que essa herdará membros de uma classe existente. Essa classe que já existe é denominada, no Java, de superclasse e a nova classe que está sendo criada é a subclasse. É importante ressaltar que cada subclasse pode tornar-se superclasse para futuras classes (DEITEL, 2005).

Uma subclasse é mais específica que a superclasse e representa, portanto, um grupo mais especializado de objetos. Em Java, a classe no topo da hierarquia de herança é a *Object*, da qual toda classe estende (direta ou indiretamente). Essa linguagem de programação suporta apenas herança simples, que ocorre quando uma classe é derivada de uma, e apenas uma, superclasse direta (DEITEL, 2005).

**2.1.2. Interfaces**

O uso de interfaces em programação orientada a objetos permite que classes não relacionadas implementem um conjunto de métodos comuns. Elas funcionam como uma documentação, especificando quais operações a classe deve realizar, mas não dizem a respeito de como a classe realizará a operação. Ao implementar determinada interface, a classe “assina” um contrato com o compilador, afirmando que vai declarar todos os métodos definidos na interface (DEITEL, 2005).

**2.2. Estruturas de dados**

Segundo Lafore (2004, p. 3), uma estrutura de dados descreve uma disposição de dados armazenados na memória de um computador (ou algumas vezes em disco). Alguns exemplos dessas estruturas são vetores, listas encadeadas, pilhas, filas e árvores binárias.

**2.2.1. Listas Encadeadas**

Uma lista encadeada ou lista vinculada é uma coleção linear, isto é, em sequência, de objetos que fazem referência para outros objetos de uma mesma classe. Esses objetos auto referenciais são chamados de nós ou células. Em geral, uma lista encadeada é acessada, inicialmente, por uma referência ao seu primeiro nó e a partir daí cada nó subsequente é acessado via referência contida no nó anterior (DEITEL, 2005).

Sendo uma estrutura de dados dinâmica, o tamanho da listas encadeadas pode aumentar ou diminuir de acordo com a necessidade do programa. Dessa forma, o seu campo de aplicação primordial é em problemas nos quais não se pode prever antecipadamente a quantidade de elementos que serão armazenados. As listas vinculadas tornam-se cheias apenas quando não há mais memória disponível no sistema para atender as solicitações de espaço da aplicação (DEITEL, 2005).

Por serem extremamente versáteis, as listas são comumente empregadas como base para diversas outras estruturas de dados, como filas e pilhas, por meio do conceito de TADs (Tipos Abstratos de Dados) (LAFORE, 2004).

**2.2.2. Filas**

Nas ciências da computação a fila é uma estrutura de dados que baseia-se no princípio FIFO (*first in, first out* ou primeiro a entrar, primeiro a sair), assim, o primeiro elemento a ser inserido na fila é também o primeiro a sair. Elas são muito semelhantes a filas encontradas no cotidiano de qualquer indivíduo, como em bancos ou em supermercados, nas quais a primeira pessoa a chegar é também a primeira a ser atendida e, consequentemente, a primeira a sair (LAFORE, 2004; DEITEL, 2005).

É fundamental notar que as filas, assim como as pilhas, são entidades conceituais, dissociadas da sua implementação. Assim, uma fila pode ser implementada tanto por um vetor, como também por uma lista encadeada. Entretanto, vale ressaltar que, em contextos nos quais não se conhece previamente a quantidade de elementos que serão armazenados, deve-se priorizar as listas encadeadas como mecanismo de implementação, pois essas fornecerão maior flexibilidade que um vetor (LAFORE, 2004).

**2.2.3. Ordenação com algoritmo *MergeSort***

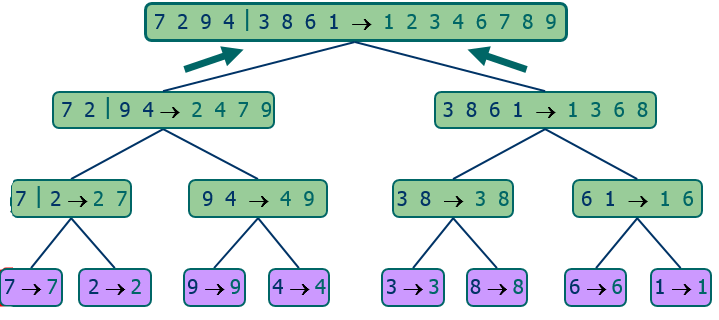
*MergeSort* é um algoritmo geralmente descrito como elegante, pois consiste em "uma ordenação que se ordena". Isso significa que o *MergeSort* requer poucas comparações e trocas, ao invés disso utiliza da estratégia de "dividir para conquistar" (WIRTH, 2009).

O primeiro passo do algoritmo é dividir a sequência a ser ordenada ao meio. Então, cada uma dessas metades obtidas são dividas ao meio novamente. O algoritmo se repete até que todas essas "subsequências" tenham exatamente um elemento. Nesse ponto, cada subsequência é ordenada. Na próxima fase do algoritmo, as subsequência são gradualmente fundidas, até que a sequência original seja obtida de forma ordenada (WIRTH, 2009).

Os exemplos de implementação mais recorrentes na literatura utilizam um método recursivo para a realização da primeira fase e um método denominado *merge* (fusão)para a realização da segunda. O custo para a execução desse algoritmo é da ordem de O(n log(n)) (WIRTH, 2009).

Na figura 1 há uma representação de como o *MergeSort* efetiva a ordenação de uma sequência de dados.

**Figura 1. Exemplo de funcionamento do *MergeSort***



**FONTE: Repositório digital da NYU *(New York University)***[[1]](#footnote-1)

**2.3. Padrões de projeto**

Gamma et al. (2000, p. 17) diz que: “Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável.”

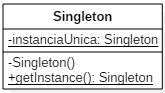
**2.3.1. Singleton**

Em determinados casos de modelagem é importante impedir a criação de mais do que uma instância de certa classe. Dessa necessidade surge o padrão *Singleton*, afim de garantir que determinada classe tenha somente uma instância e fornecer um ponto de acesso global para a mesma (GAMMA et al., 2000).

A solução geral e mais recorrente na literatura para implementação do padrão consiste em tornar a própria classe responsável pela criação e controle da sua instância única. Assim, a classe pode garantir que nenhuma outra instância seja criada, bem como fornecer um meio para acessar a instância única (GAMMA et al., 2000).

Na figura 2 há a representação da estrutura de uma classe que implementa o padrão *Singleton*.

**Figura 2. Estrutura de uma classe que implementa o padrão *Singleton***



**FONTE: Próprio autor**

**2.3.2. MVC *(Model-View-Controller)***

O princípio do padrão MVC constitui-se em dissociar o código fonte relacionado ao acesso de dados e lógica de negócios daquele responsável por gerenciar a maneira pela qual as informações são exibidas ao usuário (ECKSTEIN, 2007).

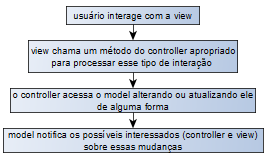
Segundo Eckstein (2007, p. 01), o MVC se divide em três camadas principais:

1. *Model* - O *model* (modelo)representa os dados e as regras que gerenciam o acesso e a atualização desses dados. Em muitas aplicações, essa camada serve também como uma aproximação em software de processos do mundo real.
2. *View* - A *view* (visão) apresenta os conteúdos do *model*. Nessa camada é especificada exatamente a maneira como os dados do *model* devem ser apresentados. Se os dados do *model* são modificados, a apresentação da *view* deve ser modificada conforme necessário.
3. *Controller* - O *controller* (controlador) traduz as interações do usuário com a *view* em ações que o *model* vai efetuar. Dependendo do contexto, o *controller* pode também selecionar uma nova *view* que será exibida para o usuário.

A relação e a comunicação entre essas camadas acontece como esquematizando em fluxograma na figura 3.

Uma das principais vantagens da utilização desse padrão consiste na possibilidade de alterações na *view* não implicarem, necessariamente, em adaptações no *controller* e no *model*. Assim, a capacidade reutilização dos projetos que implementam esse padrão é larga. Ademais, um mesmo *controller* pode gerenciar diversas *views*, possibilitando, por exemplo, que uma mesma aplicação seja apresentada de diferentes formas e em diferentes plataformas (ECKSTEIN, 2007).

**Figura 3. Comunicação entre camadas no padrão MVC**



**FONTE: Próprio autor**

**2.3.3. Padrão iterador**

Um iterador permite percorrer uma coleção de dados sem, necessariamente, conhecer a implementação interna dessa coleção. Dessa forma, é possível utilizar o iterador da mesma forma, tanto para percorrer tanto uma lista encadeada, quanto uma pilha, por exemplo. Uma das outras possibilidades capacitadas por um iterador é remover elementos de uma coleção enquanto ela é percorrida (LAFORE, 2004).

**2.4. Tratamento de exceções**

Uma exceção é a indicação da ocorrência de um problema em tempo de execução do programa. O mecanismo para tratamento desses erros permite que os programadores removam da parte “principal” do código, linhas destinadas ao gerenciamento dessas situações atípicas, de forma a potencializar a clareza do programa (DEITEL, 2005).

Em Java, o modelo para tratamento de exceções inclui a instrução *try*, o comando *throw* e a cláusula *throws*. A instrução *try* é composta pelos blocos *try*, *catch* e *finally*. No bloco *try* é inserida a parte do código que pode resultar no lançamento de uma exceção, ele deve ser seguido por um bloco *catch*, que é onde a possível exceção lançada é capturada. O bloco *finally* é opcional e as instruções contidas nele são executadas independentemente do que ocorrer nos blocos *try/catch*. Já o comando *throw* é responsável pelo lançamento de uma exceção, entretanto, o método que o contém deve, obrigatoriamente, informar o possível lançamento através da cláusula *throws* (DEITEL, 2005; GOSLIN et al., 2015).

Ainda falando sobre Java, nessa linguagem de programação todas as classes de exceções herdam direta ou indiretamente da superclasse *Throwable.* Existem, ainda, duas subclasses principais de *Throwable*: *Exception* e *Error*. Todas as exceções que herdam de *Expection*, exceto *RunTimeException*, são definidas como *checked* (checadas)e delas, geralmente, são lançadas exceções das quais os programas podem se recuperar em tempo de execução. Já as exceções geradas por *Error* e *RunTimeException* são definidas como *unchecked* (não checadas) e dessas os programas dificilmente podem se recuperar, pois refletem, respectivamente, quebra de pré-condições durante o tempo de execução e erros que estão fora do escopo do programa (memória indisponível e falhas na JVM, por exemplo) (DEITEL, 2005; GOSLIN et al., 2015).

Nas linguagens de programação que não suportam tratamento de exceções, o manuseio de erros é, geralmente, mais longo e muitas vezes alguns desses problemas podem passar despercebidos. Tudo isso resulta em softwares menos robustos (DEITEL, 2005).

**2.5. Testes unitários**

A realização de testes é fundamental no contexto de desenvolvimento de softwares. Dependendo do modelo de projeto seguido, os testes podem ser construídos antes ou depois da implementação de uma aplicação. Entretanto, ao testar o funcionamento de várias unidades de um programa ao mesmo tempo, erros podem passar despercebidos e, além disso, torna-se mais difícil encontrar um determinado problema quando várias estruturas estão sendo executadas concomitantemente.

Por outro lado, os testes unitários ou testes de unidade, baseiam-se no princípio de testar cada pequena unidade de um programa de forma completamente independente e dissociada. Assim, erros podem ser descobertos de forma mais rápida e a confiabilidade do processo de testes é potencializada.

**2.5.1. Framework JUnit para realização de testes unitários**

JUnit é um framework utilizado para execução de testes de unidade na linguagem de programação Java. Por princípio, testes realizados com JUnit visam verificar, separadamente, o funcionamento de cada unidade de um programa (STEGEMAN, 2010).

Para a realização desses testes é criada uma classe com seus respectivos métodos de teste. Nem sempre há uma correspondência unívoca entre os métodos da classe a ser testada e os métodos da classe de teste, pois podem ser criados testes para a execução satisfatória de um método, bem como testes para verificação de erros esperados em algumas situações, por exemplo. Comandos *assertion* (assertivas) são incluídos nos métodos de teste para checar se os resultados das funcionalidades ocorrem como esperado.

**3. Metodologia**

Para resolver o problema proposto inicialmente foi necessária uma leitura crítica do descritivo fornecido. Nessa leitura buscou-se definir e entender os requisitos básicos do problema (materializados pelas *UsersStories*) e quais os aspectos teóricos seriam fundamentais no desenvolvimento da solução. Alguns desses aspectos eram desconhecidos, assim, surgiu a necessidade de uma revisão bibliográfica sobre certos conceitos e eixos temáticos. Essa revisão foi realizada, principalmente, por meio da leitura de livros e artigos.

Além disso, duas reuniões semanais com um grupo tutorial foram fundamentais para debater dúvidas que surgiram apenas durante a fase de modelagem e implementação. Nessas sessões novas ideias e caminhos de resolução foram levantados e tornaram-se parâmetro fundamental para agilizar o processo de concepção.

A aplicação foi criada com base linguagem de programação Java, por meio do ambiente de desenvolvimento (IDE) *Eclipse*, na sua versão *Mars*. Ademais, todo processo de programação ocorreu na versão 8.0 do sistema operacional *Windows*.

**3.1. Modelagem do problema**

Apesar de haver uma modelagem inicial do programa, essa foi sendo modificada durante o processo de implementação, para adequação aos testes fornecidos. A modelagem do software aconteceu com base no MVC, por conta de boas práticas de projeto proporcionadas por esse padrão.

**3.1.1. Decisões de projeto**

Como pode ser observado no diagrama parcial da figura 4, a classe “CompradorFan” herda da classe “Comprador”. Inicialmente pensou-se em utilizar uma variável do tipo *boolean* para indicar se um “Comprador” era ou não fã. Entretanto, como o avanço dos estudos percebeu-se que a herança se encaixaria mais adequadamente nesse caso, já que um “CompradorFan” tem todos os atributos de um “Comprador” e mais um atributo que é o número de registro no fã clube, ou seja, um “CompradorFan” é um “Comprador”. Além disso, está explicito também que a classe “Comprador” implementa uma interface chamada “Comparável”, essa relação foi necessária para atender ao requisito de ordenação em ordem alfabética dos nomes dos compradores. Esse aspecto será explanado com mais detalhes em seguida.

Outra decisão de projeto importante foi a criação da classe “Venda”. Percebeu-se, durante o processo de desenvolvimento, que apenas a classe “Ingresso” não seria capaz de armazenar informações importantes de uma venda, como o comprador, a quantidade de ingressos vendidos e o valor total da venda. Existiam outras possíveis resoluções para essa questão, entretanto, essa foi a solução considerada mais viável.

Vale ressaltar que os métodos das classes, bem como alguns relacionamentos de dependência, foram abstraídos do diagrama apresentado na figura 4 em detrimento de uma maior organização.

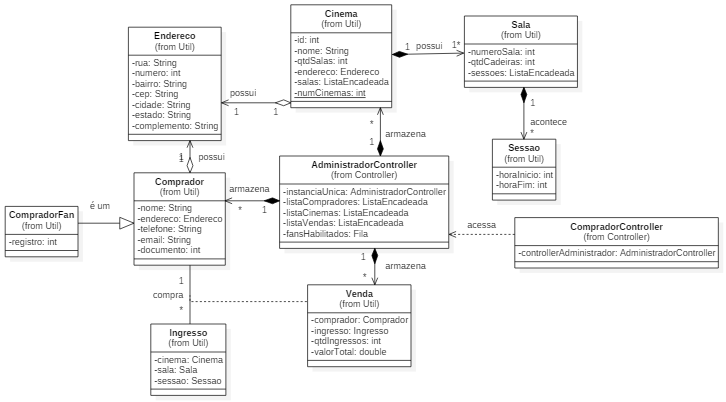
**3.2. Gerenciamento de informações**

As classes “AdministradorController”e “CompradorController”, evidenciadas na figura 4,são responsáveis por intermediar a entrada de informações no programa, sendo que essa última corresponde a uma classe de negócios, pois é responsável apenas por controlar o processo de venda de ingressos. Já em “AdministradorController”estão os métodos que realizam o gerenciamento (cadastro, remoção, alteração e listagem) dos dados referentes as classes que fazem parte do *Model*. É importante, ainda, ressaltar que ambas as classes que formam o *Controller* implementam o padrão *Singleton*, pois é fundamental garantir a unicidade delas no sistema. Esse aspecto é ainda mais importante tratando-se de “AdministradorController”, pois ele funciona também como uma espécie de banco de dados do sistema, armazenando todas as informações importantes em tempo de execução.

Contudo, as informações armazenadas no “AdministradorController” não podem ser manuseadas deliberadamente. Todas as informações registradas podem ser alteradas, exceto as vendas. O sistema também não permite a remoção total de uma venda, bem como de uma sala ou sessão específica. Cadastros de compradores e cinemas podem ser removidos, desde que esses não estejam envolvidos em uma venda.

Ainda tratando-se do gerenciamento de informações, um caso digno de destaque é o cadastro e alteração de uma sessão, pois deve haver um intervalo mínimo de 3 horas entre duas sessões cadastradas no sistema. Para garantir esse requisito foi criado um método de verificação. Esse método impede o cadastro ou alteração caso o valor absoluto (módulo) da diferença entre a hora de início informada e a hora de início de alguma sessão anteriormente cadastrada for menor que 3. Assim, garante-se tanto um intervalo de 3 ou mais horas anteriores e posteriores*.*

**Figura 4. Diagrama com as classes relacionadas à modelagem do problema.**

****

**FONTE: Próprio autor**

**3.3. Estruturas de armazenamento e organização de dados**

Apesar do problema não exigir persistência dos dados em memória, foi necessário manter eles em tempo de execução do programa. Dessa forma, foi fundamental definir uma estrutura de dados para realizar esse armazenamento.

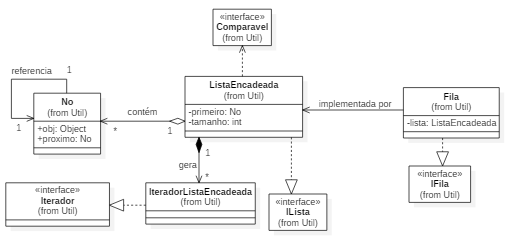
Como pode ser visualizado no diagrama das classes envolvidas na estruturação dos dados do programa, representado na figura 5, a estrutura de dados principal é a lista encadeada. Essa definição se deu por conta da natureza dinâmica das listas, elas garantem o armazenamento de uma quantidade de dados definida apenas em tempo de execução. Além disso, a trivialidade da execução de tarefas como busca e remoção numa lista corroboraram para essa escolha.

Algumas interfaces foram fornecidas para auxiliar na implementação da lista encadeada. Outra interface fornecida foi a interface “Iterador”, através dela foi implementada a classe “IteradorListaEncadeada”, responsável por gerar um objeto iterador para facilitar o processo de percorrer uma lista encadeada ou qualquer outra estrutura de dados que a use como TAD.

**3.3.1. Fila como mecanismo para distribuição de camisas**

Os compradores cadastrados no fã clube da franquia concorrem a uma camisa a cada vez que realizam uma compra de ingresso. A ordem do sorteio das camisas é com base na ordem em que a venda foi realizada, assim quem compra primeiro deve ter prioridade. A estrutura de dados mais adequada para realizar essa tarefa é a fila, por conta do seu princípio FIFO. Conforme descrito no diagrama da figura 5, a fila foi implementada utilizando a lista encadeada como TAD.

**Figura 5. Representação em diagrama das classes envolvidas na estruturação de dados**



**FONTE: Próprio autor**

**3.3.2. Ordenação de compradores**

Foi solicitado a disponibilização de uma listagem dos compradores cadastrados no sistema, ordenados segundo a ordem alfabética dos seus nomes. Essa ordenação deveria ser implementada com um algoritmo cujo custo fosse O(n log(n)). Haviam duas possibilidades de algoritmos que atendiam a esse requisito, o *QuickSort* e o *MergeSort*. Entretanto, em alguns casos o custo do *QuickSort* degenera para O(n²), então decidiu-se implementar o *MergeSort*.

Por conta da lista encadeada do sistema ser genérica foi necessário criar alguns critérios para a realização da ordenação. Só é possível realizar a ordenação de uma lista com objetos que implementem a interface “Comparavel”. Nessa interface está definido o método “comparacao”, que retorna -1, 1 ou 0, caso o objeto seja comparado com outro objeto maior, menor ou igual a ele. A definição objeto maior, menor ou igual é feita segundo critérios estabelecidos durante a implementação do método “comparação” em uma classe. No caso dos compradores, por exemplo, um objeto é maior que o outro se o seu nome for alfabeticamente maior.

**4. Resultados**

**4.1. *UsersStories* implementadas**

Todas as UsersStories sugeridas no problema foram implementadas com sucesso. Elas estão expostas na tabela 1. É importante ressaltar que a *User Story* de número 11, Recuperar sessão, não estava incluída no projeto original, mas foi criada por se mostrar bastante útil no contexto do programa.

**Tabela 1. UsersStories efetivamente implementadas**

|  |  |
| --- | --- |
| ***UsersStories*** | |
| 1 – Cadastrar cinema | 11 – Recuperar sessão |
| 2 – Alterar cinema | 12 – Listar sessões |
| 3 – Remover cinema | 13 – Cadastrar comprador |
| 4 – Listar cinemas | 14 – Alterar comprador |
| 5 – Recuperar cinema | 15 – Remover comprador |
| 6 – Cadastrar salas | 16 – Listar compradores |
| 7 – Alterar sala | 17 – Recuperar comprador |
| 8 – Listar salas | 18 – Listar compradores que concorrem a camisa |
| 9 – Cadastrar sessão | 19 – Distribuir camisas |
| 10 – Alterar sessão |  |

FONTE: Próprio autor

**4.2. Tratamento de erros de execução**

Resolver possíveis erros de execução é um dos maiores desafios no desenvolvimento de softwares. No caso do programa implementado utilizou-se o mecanismo para tratamento de exceções intrínseco à linguagem Java. Apesar de já existirem classes de exceções prontas, foi solicitado a criação de novas exceções específicas ao cenário do problema proposto. Essas classes criadas estão listadas na tabela 2.

**Tabela 2. Classes de exceções criadas especificamente para o contexto do problema**

|  |  |
| --- | --- |
| **Classes de exceção criadas** | |
| 1 – CampoObrigatorioInexistenteException | 9 – SessaoNulaException |
| 2 – CinemaNaoEncontradoException | 10 – FanHabilitadoInexistenteException |
| 3 – CompradorNaoEncontradoException | 11 – HorarioImproprioException |
| 4 – SalaNaoEncontradaException | 12 – IntervaloMinimoInsuficienteException |
| 5 – SessaoNaoEncontradaException | 13 – LimiteIngressosExcedidoException |
| 6- CinemaNuloException | 14 – LimiteSalasExcedidoException |
| 7 – CompradorNuloException | 15 – RemocaoNaoPermitidaException |
| 8 – SalaNulaException |  |

FONTE: Próprio autor

**4.2.1. Criação adicional da exceção “HorarioImproprioException”**

A criação da classe de número 11 não foi solicitada explicitamente nos requisitos do problema. Entretanto, para fins de desenvolvimento, definiu-se que um dia útil se inicia às 1:00 h e termina às 23:00 h. O mecanismo utilizado para impedir o registro de uma sessão fora desses padrões foi, justamente, o lançamento de uma exceção do tipo “HorarioImproprioException”.

**4.3. Realização de testes unitários**

No projeto original foram fornecidos testes de unidade prontos. Entretanto, ao longo do processo de desenvolvimento percebeu-se que alguns desses estavam incorretos e que possíveis casos de teste estavam sendo negligenciados. Assim, para além da correção de erros nos existentes, foram criados novos testes. Com as correções realizadas todos os testes funcionaram corretamente, elevando o nível de confiabilidade do programa.

**4.3.1. Modificações nos testes de alteração de cadastros**

Inicialmente os testes de alteração dos cadastros de qualquer entidade do sistema (“Comprador”, “CompradorFan”, “Cinema”, “Sala” e “Sessão), estavam sendo realizados de forma ineficiente, pois o próprio teste estava obtendo a referência do objeto desejado e realizando a modificação. Assim, na nova versão dos testes, todas as novas informações são passadas por argumentando para o método de alteração. Além disso, a situação na qual o usuário tenta realizar a modificação de uma entidade não cadastrada não estava sendo considerada. Para suprir essa necessidade foram criados blocos *catch* para as respectivas exceções de cadastro não encontrado e novos testes, bem como testes para verificar se as situações na qual é realizada uma tentativa de alteração de um cadastro não existente.

Como consequência dessas atualizações, os blocos *catch* para os casos em que se tentava alterar um objeto nulo, assim como os testes de alteração de um objeto nulo foram removidos. Eles tornaram-se inadequados ao novo contexto, pois não está sendo realizada uma passagem da referência de um objeto por argumento para os métodos de alteração.

**4.3.2. Modificações nos testes de listagem**

Nos testes de listagem originais estava sendo considerado que o método responsável por uma listagem retornava um inteiro condizente com a quantidade de itens cadastrados. Entretanto, esse comportamento é inadequado. Dessa forma, decidiu-se retornar um iterador nos métodos de listagem e para garantir que a quantidade de itens percorridos por esse iterador é equivalente à quantidade de elementos cadastrados um contador é incrementado enquanto um próximo registro é encontrado.

**4.3.3. Modificações nos testes de distribuição de camisas**

O princípio básico de testar apenas uma unidade por vez estava sendo ignorado no teste original de distribuição de camisas, de forma que o mesmo método de teste era responsável por verificar os casos de sucesso e erro na distribuição. Assim, foi fundamental dividi-lo em dois testes, um no qual o processo bem sucedido é garantido e outro para verificar o caso em que erros são gerados. Ademais, foi necessário acrescentar um bloco *catch* às chamadas do método de distribuição de camisas, pois não estava sendo considerado que ele pode lançar uma exceção caso não existam mais fãs aptos a receber o prêmio.

**4.3.3. Elaboração de teste para o *MergeSort***

Apesar de o processo de ordenação como um todo ser testado no teste de listagem de compradores, não havia uma verificação do funcionamento de cada uma das unidades que formam o algoritmo *MergeSort*. Assim, foram criados testes para garantir que os passos de recursão e fusão (*Merge*) são executados com sucesso, tanto de forma dissociada, quanto em conjunto.

**5. Conclusão**

O negligenciamento do controle de informações em um estabelecimento comercial pode culminar em prejuízos incomensuráveis para empresa. Isso torna-se ainda mais grave ainda quando a quantidade de dados a ser gerenciada é grande, a exemplo do que acontece no lançamento de um novo filme da franquia Star Wars, um acontecimento de proporções mundiais. Dessa forma, é fundamental um sistema informatizado, como o que foi descrito nesse relatório, para processar e administrar essa grande quantidade de dados.

O projeto concebido atende a todos os requisitos solicitados no problema. Pode-se garantir ainda que todos os métodos implementados estão funcionando como esperado, pois o resultado de todos os testes fornecidos foi positivo, assim como o resultados dos testes criados durante o desenvolvimento. A utilização do padrão MVC colaborou para a construção de um projeto organizado e com grande potencial de reutilização.

Contudo, a interface gráfica, representada pela camada de visão *(view)* não foi criada. Atualizações posteriores do software podem ser no sentido de inclusão dessa camada, assim, a aplicação será acessível também para usuários que não entendem de conceitos relacionados ao processo de programação. Além disso, outro possível avanço no sistema seria a integração de um banco de dados, de forma a potencializar a capacidade de armazenamento de registros.

Pode-se observar, portanto, que, com base no que foi proposto, a implementação do projeto foi extremamente favorável. Principalmente por ter servido como forma de consolidação e aplicação prática dos conhecimentos teóricos das disciplinas que compõem o módulo integrador, além da introdução de alguns aspectos relacionados à engenharia de software.

**6. Referências**

BOOCH, G. et al. **Object-Oriented Analysis and Design with Applications.** 3 ed. Boston: Pearson Education, 2007.

DEITEL, H.M.; DEITEL, P.J. **Java – Como Programar.** 6 ed. São Paulo: Pearson Prentice Hall, 2005.

ECKSTEIN, R. **Java SE Application Design With MVC.** Disponível em: <http://www.oracle.com/technetwork/articles/javase/index-142890.html>. Acesso em: 06 mar. 2016.

GAMMA, E. et al. **Padrões de Projeto – Soluções reutilizáveis de software orientado a objetos.** Porto Alegre: Bookman, 2000.

GOSLING, J. et al. **The Java® Language Specification - Java SE 8 Edition.** California, v8, fev. 2013.

LAFORE, R. **Estruturas de dados & Algoritmos em Java.** Rio de Janeiro: Editora Ciência Moderna Ltda., 2004.

STEGEMAN, J. **Testing Your Application with JUnit.** Disponível em: < http://www.oracle.com/technetwork/articles/adf/part5-083468.html>. Acesso em: 06 mar. 2016.

WIRTH. N. **Algoritmos e Estruturas de Dados.** Rio de Janeiro: LTC, 2009.

1. Disponível em: http://www.cs.nyu.edu/courses/fall06/V22.0102-001/lectures/mergesort-102-fa06.ppt. Acesso em mar. 2016. [↑](#footnote-ref-1)