**Relatório do Problema 3: “Software para Compactação e Descompactação de Arquivos”**

**Valmir Vinícius de Almeida Santos**

Engenharia de Computação – Universidade Estadual de Feira de Santana  
Feira de Santana, 10 de abril de 2016

vvalmeida96@gmail.com

**1. Introdução**

Compressão e descompressão são atividades fundamentais no escopo de estudos relacionados ao gerenciamento de dados. Inicialmente, essa importância surge atrelada aos altos custos envolvidos no armazenamento de informações, afinal as mídias possuíam capacidade extremamente limitada. Com o passar do tempo, esse aspecto tornou-se secundário e, atualmente, a compactação vem sendo utilizada com a finalidade de minimizar o tempo gasto na transmissão de dados em redes virtuais.

Esse relatório visa descrever e detalhar o processo de modelagem e implementação, de um compactador e descompactador de arquivos dos tipos: *.txt*, *.html*, *.c* e *.cpp*, denominado de *WINMonster*. O software desenvolvido realiza, ainda, a verificação da integridade dos arquivos durante a descompactação. Além disso, foi construída uma interface gráfica, composta por botões, caixas de texto, menus e outros componentes visuais.

O projeto foi concebido por meio da linguagem de programação Java, seguindo os conceitos do paradigma de programação orientada a objetos e o padrão de projeto MVC *(Model-View-Controller).* O método de *Huffman* foi o mecanismo implementado para realizar a codificação de dados, sendo que o seu funcionamento está atrelado a estruturas de dados auxiliares, como árvore binária, fila de prioridade e lista encadeada. Ademais, buscando-se potencializar a robustez do software, foram desenvolvidos testes unitários. As telas do sistema foram construídas por meio das classes disponibilizadas pela biblioteca *Swing*.

**2. Fundamentação teórica**

**2.1. Estruturas de dados**

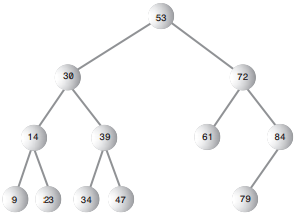
**2.1.1. Árvores**

Árvore é um tipo de estrutura de dados que consiste de nós (ou células) interligados por arestas (linhas), conforme representado na figura 1. Nesses nós estão contidas as entidades do programa que se deseja armazenar em tempo de execução e a única maneira de acessá-las é seguindo o caminho das linhas, ou seja, indo de nó em nó (LAFORE, 2004).

A definição de árvore está diretamente ligada ao conceito de raiz. A raiz é um nó superior e único, a partir do qual se dá o acesso aos outros nós. Além da raiz, outros nós possuem nomenclaturas especiais. É definido como pai, qualquer nó (exceto a raiz), que possui uma ou mais arestas em direção a outros nós, que por sua vez são chamados de filhos. Por outro lado, um nó que não possui filhos é denominado de folha. Uma árvore pode possuir diversas folhas (LAFORE, 2004).

A cada nó de uma árvore é associado um valor chave, que é utilizado como mecanismo de localização. Para percorrer todos os nós são empregados, geralmente, métodos recursivos (LAFORE, 2004).

**Figura 1. Representação de uma árvore.**



**FONTE: (LAFORE, p. 339, 2005)**

**2.1.1.1. Árvore binária**

No caso em que todo nó possui no máximo dois filhos a árvore é classificada como binária. Nesse tipo de estrutura cada filho é chamado de “filho à esquerda” ou “filho à direita”. Contudo, os nós em uma árvore binária não têm necessariamente dois filhos, elas podem ter apenas um deles ou ainda não ter nenhum (neste caso, é uma folha) (LAFORE, 2004).

**2.1.2. Filas de prioridade**

Uma fila de prioridade é uma estrutura de dados que possui aplicações mais especializadas do que uma pilha ou uma fila. Como acontece numa fila comum, existe uma frente e um fundo, de onde os itens são removidos. Entretanto, na fila de prioridade, os dados são ordenados segundo um valor chave especificado, de modo que aqueles com maior prioridade estão sempre nas posições iniciais. Para manter essa organização, os itens são inseridos ordenadamente (LAFORE, 2004).

É importante ressaltar que a definição de maior ou menor prioridade varia de acordo com as especificidades do problema no qual a estrutura é empregada. Dessa forma, itens com chaves mais altas podem estar localizados tanto na frente quanto no fundo, o que acontece também para os itens com chaves mais baixas (LAFORE, 2004).

**2.3. Codificação de Huffman**

Desenvolvido por *David Huffman*, em 1952, a codificação de *Huffman* é um dos métodos mais conhecidos para codificar dados. O maior campo de aplicação desse algoritmo é para a compactação de textos. No artigo em que publicou a sua invenção, *David Huffman* a definiu como: “um método para a construção de códigos com o mínimo de redundância”, o que resume o propósito do algoritmo (LAFORE, 2004).

**2.3.1. Códigos de caracteres**

Em um texto normal cada caractere é representado por um tamanho fixo de bytes e, consequentemente, por um mesmo número de bits. Desse modo, independente da frequência em que ocorrem no texto, os caracteres possuem a mesma magnitude. A base da codificação de *Huffman* está, exatamente, em representar aqueles caracteres com maior taxa de repetição através de um código com quantidade reduzida de bits, enquanto aqueles que aparecem com maior frequência podem ser representados com códigos maiores (LAFORE, 2004).

Entretanto, para que em uma posterior decodificação não ocorram ambiguidades, é fundamental seguir a regra: “nenhum código pode ser prefixo de qualquer outro código”. Supondo que em um caso hipotético essa regra fosse desconsiderada, a letra A poderia ser representada pelo código 01 e a letra B por 0110. Desse jeito, não saberíamos se uma sequência 01 significa, na verdade, o código da letra A ou os dois primeiros bits do código de B (LAFORE, 2004).

**2.3.2. Geração de códigos**

Para a criação dos códigos é fundamental implementar um tipo de árvore binária especial, denominada de árvore de *Huffman*. Nessa árvore, cada nó folha armazena um dos caracteres do texto a ser codificado e a sua respectiva frequência, de forma que aqueles que quanto mais alta for a frequência do caractere, mais alto ele aparecerá na árvore. Consequentemente, os outros nós armazenam apenas a soma das frequências dos seus filhos (LAFORE, 2004).

Para obter o código de um caractere é necessário percorrer a árvore a partir da raiz até a folha que o contém. Durante esse caminho, a cada ida para um nó à direita deve ser adicionado um bit 1 ao código e a cada ida à esquerda deve ser adicionado um bit 0. (LAFORE, 2004).

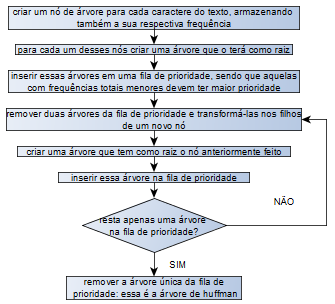
**2.3.3. Criação da árvore de Huffman**

Existem diversas abordagens utilizadas para a criação da árvore de Huffman. Uma delas é baseada no emprego de filas de prioridade e segue o algoritmo que está representado em fluxograma na figura 2.

**2.3.4. Decodificação**

Para obter o texto original a partir de uma mensagem codificada é fundamental armazenar a árvore ou o dicionário construído. Dessa maneira é possível obter a representação unívoca em caractere de cada código (LAFORE, 2004).

**Figura 2. Algoritmo para criação da árvore de Huffman**



**FONTE: Próprio Autor**

**2.4. Construção de interfaces gráficas em Java**

**2.4.1. *Look and feel***

*"Look and feel"* em Java é a interface padrão para aplicações construídas com as bibliotecas intrínsecas à linguagem de programação. *“Look”* refere-se a aparência da aplicação, enquanto *“feel”* ao seu comportamento. Esse modelo foi concebido de forma a trazer para as interfaces gráficas o comportamento multiplataforma do Java, ou seja, garantir uma consistência na aparência e no comportamento nos elementos de *design*. Entretanto, algumas bibliotecas, como a *Swing*, permitem trocar o *“Look and feel*” padrão para um outro estilo que o programador desejar (WESLEY, 2001).

**2.4.2. Padrão *Component/Container***

Java implementa alguns padrões nas bibliotecas utilizadas para criação de interfaces gráficas. O principal deles é baseado nas classes *Component* e *Container*. O princípio básico utilizado é que cada parte constituinte da interface (botões, telas e outros), pode se comportar como um componente ou como um agrupamento de outros componentes (*container*). Dessa forma, é possível acrescentar botões em janelas e submenus dentro de menus, por exemplo (WESLEY, 2001).

**2.4.3. Orientação a eventos**

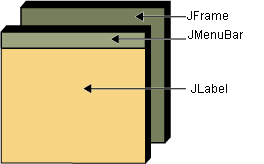
A interface gráfica em Java é definida como orientada a eventos. Cada interação do usuário com algum dos itens da *GUI* (*Graphical User Interface* – Interface gráfica do usuário) resulta em um evento, que pode ou não ser gerenciado. Caso o programador deseje que uma interação com um determinando item resulte em um evento, deve adicionar um *listener* (ouvinte) a ele. Um *listener* (ouvinte) é uma classe criada, especificadamente, para definir as ações decorrentes de um evento. Os eventos podem ser classificados de acordo com a sua origem, dessa forma existem eventos resultantes de interação com mouse, teclado, etc. (WESLEY, 2001)

**2.4.4. Biblioteca *Swing***

*Swing* é uma das mais conhecidas e utilizadas bibliotecas para a criação de interfaces gráficas em Java. Algumas das principais classes fornecidas por essa biblioteca são: *JButton* (botões), *JMenu* (menus), *JLabel* (área para exibição de textos e imagens), *JPanel* (painel), *JTextArea* (caixa de texto), dentre outros. (WESLEY, 2001)

Entretanto, para ser exibido é fundamental que esses componentes, ou suas composições, estejam dentro de um *container* raiz. Esse *container* é o topo da hierarquia formada por todos os componentes *Swing* utilizados em uma interface gráfica. O *JFrame* é um dos *containers* raiz mais utilizados. Na figura 3 há um exemplo de janela formada por *JFrame* e outros componentes da *Swing.*[[1]](#footnote-1)

**Figura 3. Janela formada por componentes Swing ¹**

**

**2.5.1. Escrita e leitura de arquivos**

O Java interpreta cada arquivo como um fluxo sequencial de *bytes*. Esses fluxos podem ser utilizados para entrada e saída de dados na forma de caracteres ou *bytes*. No primeiro caso, os arquivos lidos ou gravados são do tipo binário, enquanto no segundo, são arquivos de texto (DEITEL, 2005).

Esse processamento de arquivos é feito com base nas classes do pacote *java.io*, que inclui as classes de fluxo. Segue abaixo uma listagem dessas classes e do tipo de fluxo associado a elas:

**1- *FileInputStream*:** entrada baseada em bytes de um arquivo.

**2- *FileOutputStream*:** saída baseada em bytes para um arquivo.

**3- *FileReader*:** entrada baseada em caracteres de um arquivo.

**4-** ***FileWriter***: saída baseada em caracteres para um arquivo.

Outra possibilidade proporcionada é a entrada ou saída de objetos ou variáveis primitivas. Internamente, esses dados ainda são armazenados como caracteres ou *bytes* nos arquivos, mas existem classes que facilitam essa tarefa, como a *ObjectInputStream* e a *ObjectOutputStream*, empregadas para a gravação e leitura, respectivamente, de objetos em arquivos binários. Os objetos a serem gravados ou lidos devem, obrigatoriamente, implementar a interface *Serializable* (DEITEL, 2005).

**2.6. Função de *Hash***

Uma função de *hash* é responsável por mapear um código, a partir da chave informada. Essas funções são definidas muitas vezes como de “mão única”, pois é trivial determinar o valor de *hash* a partir da chave, mas é matematicamente impossível (ou extremamente difícil), realizar a operação inversa.

Entretanto, essas funções podem resultar em colisões. Uma colisão acontece quando, para uma ou mais chave diferentes, são obtidos o mesmo valor de *hash*. Esse parâmetro é utilizado na avaliação de robustez da função, de forma que a robustez é potencializada quando a probabilidade de se ocorrer uma colisão é baixa.

As funções de *hash* possuem um campo de aplicação bem diversificado. Os exemplos mais comuns de aplicação são para a determinação de índices em uma tabela *hash*, verificação da integridade de arquivos e na área da criptografia.

**3. Metodologia**

Inicialmente, foi fundamental entender a conjuntura do problema para iniciar a sua resolução. Com a leitura crítica do texto proposto, tornou-se possível delimitar quais seriam os aspectos teóricos que deveriam ser estudados e compreendidos antes de iniciar, propriamente, o processo de modelagem e implementação.

Além disso, durante o processo de desenvolvimento, as sessões com o grupo tutorial foram imprescindíveis, pois nessas surgiram novas ideias para a implementação e as principais dúvidas foram debatidas e solucionadas. Tendo sido implementado em dupla, existiram, ainda, discussões internas sobre a modelagem do problema, buscando um formato que, além de atender aos requisitos, culminasse em um código fonte capaz de seguir as melhores práticas de programação

A aplicação foi construída utilizando-se a linguagem de programação Java e o ambiente de desenvolvimento integrado *Netbeans*, na sua versão 8.1. Por existirem dois programadores atualizando um mesmo projeto concomitantemente, foi fundamental utilizar o *GitHub*, uma plataforma para controle e gerenciamento de versões.

**3.1. Modelagem do problema**

A modelagem do projeto teve como direcionador o padrão MVC, que foi responsável por potencializar o uso das boas práticas de programação.

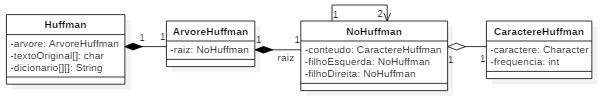
**3.1.1. Estruturação do Algoritmo de *Huffman***

Existem diversas abordagens possíveis para a implementação da codificação de *Huffman*. As estruturas de dados envolvidas, por exemplo, podem variar de uma implementação para outra. Entretanto, o resultado obtido é semelhante e válido em todas as formas de aplicar o algoritmo.

No caso do projeto desenvolvido, o algoritmo é, efetivamente, implementado, na classe “*Huffman*”. A maioria dos métodos dessa classe são privados, pois correspondem a atividades que não devem ser acessadas e estar disponíveis aos utilizadores da classe. Os únicos métodos disponíveis correspondem, exatamente, aos que realizam a codificação e a decodificação.

Uma primeira abordagem de implementação consistia em guardar cada caractere e a sua respectiva frequência em instâncias da classe “NoHuffman”, que corresponde a um nó da árvore de *Huffman*. Contudo, buscando-se potencializar a orientação a objetos da implementação, foi criada a classe “CaractereHuffman”, que armazena unicamente o caractere e a sua frequência. Essas classes e todas as outras relacionadas à codificação de *Huffman* estão representadas no diagrama de classes da figura 5. Os métodos das classes foram suprimidos, para proporcionar uma percepção potencializada da modelagem.

**Figura 5. Modelagem das classes implementadas para o algoritmo de *Huffman***



**FONTE: Próprio Autor**

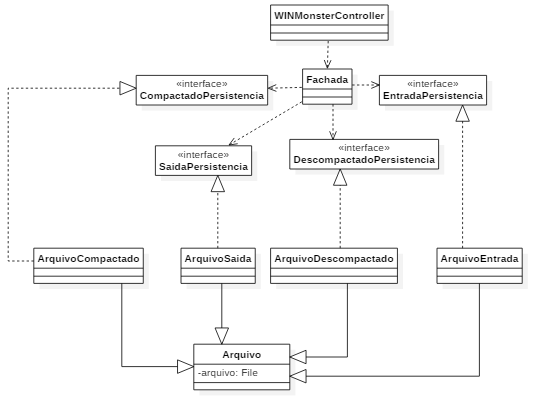
**3.1.2. Entidades de persistência**

Apesar de não ser um requisito do problema, foi decidido que a camada de negócios seria dissociada com o mecanismo específico (arquivo) de entrada e saída de dados. Dessa forma, caso o projeto fosse modificado para uma conexão com banco de dados, por exemplo, não seria necessário realizar alterações no *Controller*.

Para materializar esse modelo, foram criadas interfaces “CompactadoPersistencia”, “DescompactadoPersistencia”, “EntradaPersistencia” e “*SaidaPersistencia*”, elas correspondem a uma entidade compactada, descompactada, de entrada e de saída, respectivamente, no contexto do programa. Cada uma delas deve ser implementada pela classe que específica o tipo de entidade utilizada, no caso desse problema as classes foram “ArquivoCompactado”, “ArquivoDescompactado”, “ArquivoEntrada” e “ArquivoSaida”.

Além disso, foi criada a classe “Fachada”, que é onde ocorre a criação de instâncias das classes relativas a entidade de entrada/saída utilizada. Assim, a camada de negócios pode solicitar da “Fachada” uma instância e trabalhar apenas com as referências de interface, não tendo uma relação direta com mecanismos específicos. Esse modelo está representado no diagrama de classes da figura 4:

**Figura 4. Diagrama das classes de persistência.**



**FONTE: Próprio Autor**

**3.2. Compactação**

O processo de compactação é realizado a partir do recebimento da localização do arquivo original. Entretanto, antes de iniciar, é necessário verificar a extensão do arquivo, pois apenas aqueles com as extensões *.txt*, *.html*, *.c* e *.cpp* podem ser compactados. Para essa verificação foi criado um método estático, que lança uma exceção “FormatoInvalidoException”, caso o arquivo não possa ser compactado pelo programa.

Após essa verificação, é realizada a codificação com o algoritmo de Huffman e o texto codificado é convertido de *String* para a sua representação em bits, por meio da classe *BitSet*. Essa representação é gravada em arquivo binário, juntamente com a árvore gerada na codificação, o número de tipos de caracteres presentes no texto, o código de *hash* gerado e nome original do arquivo. A saída possui extensão *“.monster”.*

**3.3. Descompactação**

Assim como no processo de compactação, é fundamental verificar a extensão do arquivo aberto para descompactação. Essa verificação é feita com o mesmo método citado anteriormente, entretanto, usa-se um parâmetro do tipo *boolean,*a fim de indicar que trata-se de uma verificação relativa a descompactação.

Todos os dados gravados no arquivo compactado são importantes para obter o arquivo original. Com a árvore de Huffman e o número de tipos de caracteres é reconstruído o dicionário utilizado na codificação. Já através do objeto *BitSet* lido, é gerada, novamente, a representação em *String* do texto codificado.

Após esse processo é possível realizar a decodificação pelo algoritmo de *Huffman* e obter, novamente, o texto original. Caso ocorra uma violação de integridade do texto, uma exceção do tipo “IntegridadeVioladaException” é lançada, caso contrário, ele é armazenado em um arquivo com mesmo nome e extensão daquele que foi inicialmente informado na compactação.

**3.3.1. Verificação de integridade**

A verificação da integridade durante a descompactação visa garantir que nenhuma informação do texto foi corrompida. Essa avaliação foi realizada por meio de uma função de *hash* que está descrita em :

Nessa expressão, é o código de hash obtido, corresponde ao valor numérico de um caractere, é o número de caracteres do texto e é a posição do último caractere na *String* que está sendo submetida à função de *hash*. A versão iterativa de , com base na sintaxe da linguagem Java, está representada na figura 6:

**Figura 6. Representação iterativa da função de *hash***



**FONTE: Próprio Autor**

**3.3.1.1. Avaliação de robustez da função de *hash***

O primeiro método para a obtenção de códigos *hash* projetado não levava em consideração a posição das letras, apenas a sua representação em valor numérico. Buscando-se aumentar a robustez da função, o parâmetro posição do caractere na *String* passou a ser levado em consideração. Entretanto, quando comparado com as técnicas complexas disponíveis no mercado, como o MD5, o método que foi implementado na solução não se mostra tão robusto e eficaz.

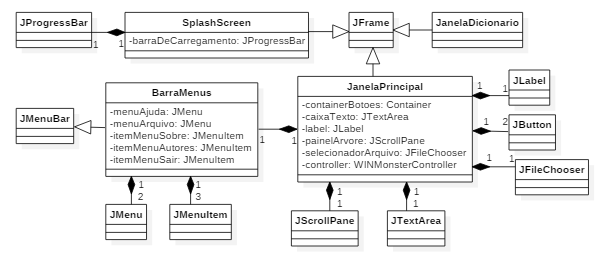
**3.4. Desenvolvimento de testes unitários**

A fim de elevar o grau de confiabilidade foram desenvolvidos testes para cada uma das principais unidades do software, por meio do framework *JUnit*. As situações consideradas nos testes correspondem aos diversos casos de uso do programa, como, por exemplo, a verificação do funcionamento correto para todos os tipos de arquivo aceitos e o lançamento de exceção durante a tentativa de abrir um arquivo com extensão imprópria. Além disso, as estruturas de dados implementadas também foram testadas. Os resultados desses testes são evidenciados e comentados na sessão seguinte.

**3.5. Construção da interface gráfica**

As entidades que formam a *View* foram montadas, basicamente, aliando as classes fornecidas pela biblioteca Swing aos conceitos de herança e composição. Dessa forma, foi possível aplicar classes já existentes na criação de janelas personalizadas, por exemplo. A modelagem das classes que fazem parte da interface gráfica está representada no diagrama de classes da figura 7.

**Figura 7. Diagrama de Classes da *View***

****

**FONTE: Próprio Autor**

**4. Resultados**

**4.1. *Users Stories* implementadas**

O texto do problema informou as *Users Stories* que, obrigatoriamente, deveriam ser implementadas. Na tabela 1 estão listadas essas *Users Stories* e o percentual de cada uma delas que foi, efetivamente, implementado.

**Tabela 1. *UsersStories* implementadas**

|  |  |
| --- | --- |
| ***User Story*** | **Percentual de Implementação** |
| 1 – Gerenciar arquivos | 100% |
| 2 – Compactar e descompactar arquivos | 100% |
| 3 – Verificar integridade | 100% |

FONTE: Próprio Autor

**4.2. Criação de testes unitários**

Os testes desenvolvidos enfocam em cada uma das principais unidades do programa de forma individual, a fim de potencializar a robustez da aplicação. Na tabela 2 estão listados os testes realizados e o percentual de funcionamento obtido a partir deles.

**Tabela 2. Testes unitários realizados e percentual de funcionamento obtido**

|  |  |
| --- | --- |
| **Teste Realizado** | **Percentual de Funcionamento** |
| 1- WINMonsterControllerTest | 100% |
| 2- ArquivoCompactadoTest | 100% |
| 3- ArquivoDescompactadoTest | 100% |
| 4- ArquivoEntradaTest | 100% |
| 5- ArquivoSaidaTest | 100% |
| 6- FilaPrioridadeTest | 100% |
| 7- HashTest | 100% |
| 8- HuffmanTest | 100% |
| 9- ListaEncadeadaTest | 100% |

FONTE: Próprio Autor

Os resultados obtidos mostram que cada unidade do software funciona, exatamente, dentro dos parâmetros esperados.

**4.3. Análise da taxa de compactação**

Objetivando-se mensurar a qualidade da aplicação desenvolvida foram realizadas comparações entre a taxa de compactação do WINMonster e de dois dos principais compactadores disponíveis no mercado atualmente: WINRar e WINZip. A análise levou em consideração três dos tipos de arquivos aceitos pelo WINMonster: .txt, .html e .cpp e, para cada um deles, quatro tamanhos diferentes de arquivos: 1 MB, 5 MB, 10 MB e 20 MB. Os resultados estão expostos nas tabelas 3, 4 e 5.

**Tabela 3. Comparação das taxas de compactação para arquivos .txt**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1 MB** | **5 MB** | **10 MB** | **20 MB** |
| **WINMonster** | 38,46% | 42,21% | 42,28% | 42,30% |
| **WINRar** | 94,77% | 97,49% | 98,68% | 99,28% |
| **WINZip** | 94,86% | 96,91% | 97,37% | 97,61% |

FONTE: Próprio autor

**Tabela 4. Comparação das taxas de compactação para arquivos .html**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1 MB** | **5 MB** | **10 MB** | **20 MB** |
| **WINMonster** | 44,3% | 44,56% | 44,69% | 44,71% |
| **WINRar** | 99,1% | 99,33% | 99,84% | 99,89% |
| **WINZip** | 98,1% | 99,36% | 98,63% | 98,67% |

FONTE: Próprio autor

**Tabela 5. Comparação das taxas de compactação para arquivos .cpp**

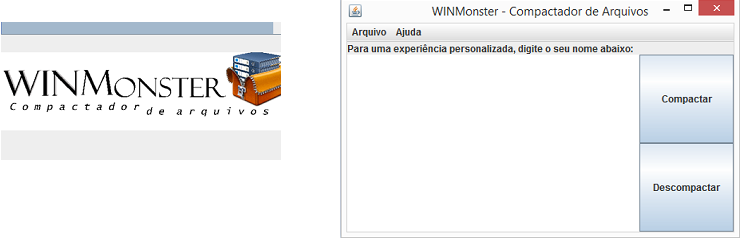
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1 MB** | **5 MB** | **10 MB** | **20 MB** |
| **WINMonster** | 38,52% | 39,06% | 39,12% | 39,14% |
| **WINRar** | 94,77% | 98,82% | 99,33% | 99,59% |
| **WINZip** | 96,17% | 98,81% | 98,82% | 99,01% |

FONTE: Próprio autor

Através desses resultados, pode-se inferir que as taxas de compactação obtidas pela aplicação desenvolvida estão abaixo de outros softwares da área. Esses resultados demonstram que as técnicas utilizadas por outras aplicações são mais complexas e robustas do que o algoritmo de *Huffman*.

**4.4. Telas do sistema**

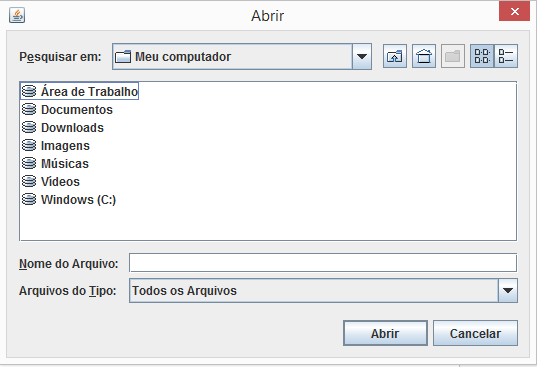
A primeira tela exibida ao iniciar a aplicação é a *Splash Screen*, que mostra ao usuário uma barra de carregamento e o logotipo do programa. Logo em seguida é apresentada a janela principal, contendo botões, menu e caixa de texto. Essas telas estão evidenciados na figura 8.

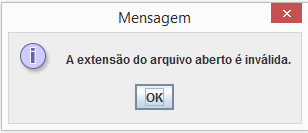
**Figura 8. *Splash Screen* e Janela Principal da Aplicação**

**FONTE: Próprio Autor**

Já na figura 9 estão representadas a janela de busca de arquivos e a caixa de mensagem exibida quando o usuário tenta compactar/descompactar um arquivo cuja extensão é inválida.

**Figura 9. Janela para Busca de Arquivos e Mensagem de Erro.**

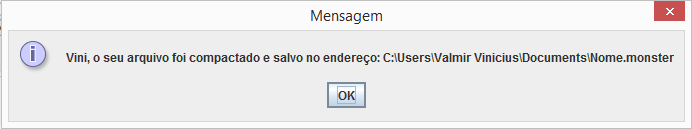




**FONTE: Próprio Autor**

A janela de mensagem da figura 10 é exibida ao final do processo de compactação. É importante ressaltar que, caso o usuário informe o seu nome na caixa de texto da janela principal ao início da aplicação, a mensagem exibida é personalizada. No processo de descompactação a mensagem exibida é similar.

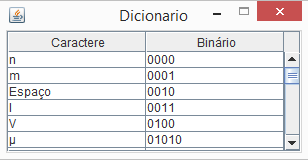
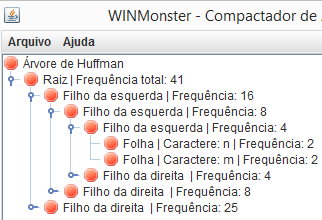
**Figura 10. Janela de Mensagem Personalizada ao Final da Compactação.**

****

**FONTE: Próprio Autor**

Na figura 11 está destacada a janela contendo o dicionário e a representação da árvore utilizados na solução. Ambos foram gerados através do algoritmo de *Huffman* e são exibidos em tela ao final do processo de compactação e descompactação.

**Figura 11. Dicionário e árvore utilizados na solução.**

**FONTE: Próprio Autor**

**5. Conclusão**

Seja para armazenamento ou transmissão, a compactação de dados mostra-se uma atividade de grande aplicação nos últimos anos, afinal há uma busca constante por dinamizar processos, diminuir tempo e custos envolvidos. Entretanto, para garantir que essas informações foram armazenados ou transmitidas de forma correta é fundamental realizar a verificação de integridade, pois só assim é possível garantir que não houve danificação total ou parcial dos elementos envolvidos.

O software aqui descrito, atende a todos requisitos mínimos no problema. Contudo, as taxas de compactação e o tempo de processamento estão aquém do que é proporcionado pelos programas disponíveis mercado. Ademais, problemas em tempo de execução surgem caso a entrada seja um arquivo de tamanho igual ou superior a 30 MB. Dessa forma, atualizações futuras podem ser focadas em resolver esses problemas e tornar, portanto, o executável mais competitivo no mercado.

Vale ainda ressaltar que, apesar de não ter sido aplicada, a filosofia de desenvolvimento orientado a testes seria capaz de tornar o processo de implementação mais eficaz e dinâmico, pois todos os possíveis erros de programação já seriam captados durante o desenvolvimento. Contudo, a construção do projeto foi de grande benefício, afinal, serviu para consolidar conhecimentos teóricos relativos ao paradigma de programação orientada a objetos, aos padrões de projeto e a criação de interfaces gráficas. Além disso, proporcionou a introdução de conceitos relacionados à engenharia de software.

**6. Referências**

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

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

WESLEY, A. **Java Look And Feel – Design Guidelines.** Boston: Addison Wesley Professional, 2001.

1. Disponível em: https://docs.oracle.com/javase/tutorial/uiswing/components/editorpane.html. Acesso em abr. 2016. [↑](#footnote-ref-1)