**UNIVERSIDADE FEDERAL DE SERGIPE**

**CIÊNCIA DA COMPUTAÇÃO**

**RAUL OLIVEIRA DE ANDRADE**

**AILTON DE ALMEIDA NETO**

**SPELLCHECKER**

**SÃO CRISTÓVÃO - SE**

**2017**

**RAUL OLIVEIRA DE ANDRADE**

**AILTON DE ALMEIDA RANGEL NETO**

**SPELLCHECKER**

Trabalho de Curso submetido à Universidade Federal de Sergipe como parte dos requisitos necessários para a aprovação na disciplina Paradigmas de Programação. Sob a orientação do Professor Doutor Adimilson de Ribamar Lima Ribeiro.

**SÃO CRISTÓVÃO- SE**

**2017**

**SUMÁRIO**

1. **INTRODUÇÃO**
2. **CONCEITOS**
   1. **VALORES E EXPRESSÕES**
   2. **TIPOS COMPOSTO E RECURSIVO**
   3. **SISTEMA DE TIPOS**
   4. **VARIÁVEIS E ARMAZENAMENTO**
   5. **VÍNCULOS E ESCOPO**
   6. **ABSTRAÇÃO PROCEDURAL**
   7. **ABSTRAÇÃO DE DADOS**
   8. **ABSTRAÇÃO GENÉRICA**
   9. **EXCEÇÕES**
3. **CONSIDERAÇÕES FINAIS**

**REFERÊNCIAS**

**RESUMO**

Nos dias atuais existem uma infinidade de linguagens de programação, próprias ou não para um determinado paradigma, cada uma com suas particularidades. Deste modo é preciso conhecer as características da linguagem usada para o desenvolvimento, bem como o paradigma que melhor se encaixa para solucionar determinado problema, explorando melhor as funcionalidades, trazendo desempenho e organização na implementação. O projeto sugerido SpeelChecker, tem como proposta a criação de uma aplicação que corrige um determinado texto , a partir de um dicionário predefinido da língua Inglesa. Sua implementação feita na linguagem de programação JAVA, seguindo os conceitos do Paradigma OO(Orientação a objetos), estimula a exploração das ferramentas contidas na linguagem e do paradigma escolhido, com base no conteúdo apresentado em sala de aula. Este documento descreve essa exploração, relatando as características relevantes no decorrer da produção da aplicação.

Palavras-chave : Linguagem de programação , aplicação, paradigma, características  
  
  
**ABSTRACT**

In the present day there are a multitude of programming languages, whether or not specific to a given paradigm, each with its own particularities. In this way it is necessary to know the characteristics of the language used for the development, as well as the paradigm that best fits to solve a certain problem, exploring better the functionalities, bringing performance and organization in the implementation. The suggested project SpeelChecker, proposes the creation of an application that corrects a certain text, from a predefined dictionary of the English language. Its implementation in the JAVA programming language, following the concepts of the OO(Object Oriented) Paradigm, stimulates the exploration of the tools contained in the language and paradigm chosen, based on the content presented in the classroom. This document describes this exploration, reporting the relevant characteristics during the production of the application.

Keywords: Programming language, application, paradigm, features

1. **INTRODUÇÃO**

A partir da compreensão do tratamento para o problema proposto, onde deve-se realizar uma leitura do texto do usuário e do dicionário, verificando se as palavras dispostas neste texto condizem com as palavras do dicionário, dando um retorno visual ao usuário, é explorado neste documento as características da linguagem e paradigma, utilizadas para o desenvolvimento da aplicação que o soluciona . Analisando estas características e necessidades, foi escolhido o paradigma OO (Orientação a objetos) , justamente pela sua organização, a maneira de representação de uma ideia/conceito em um objeto abstrato com características e ações do mundo real, além da facilidade de manutenção e leitura do código, ou seja robustez, adaptabilidade e reusabilidade.

Partindo dessa primeira escolha do paradigma, foi verificado então as LP’s que utilizam tal, e dentre as opções (C++, Java e C#) foi escolhido o Java por mais se adequar os princípios de OO :  abstração, encapsulamento e modularidade, com ferramentas que trazem facilidade ao programador, tanto no uso da linguagem em si, quanto na aplicação do paradigma escolhido, analisando também o fato de que a aplicação não necessita de um desempenho avançado, visto o escopo do problema. Sabe-se que Java também suporta o paradigma imperativo, mas foi decidido não abordá -lo nesta aplicação.

1. **CONCEITOS**

Iniciamos a produção da aplicação “SPELLCHECKER”, seguindo a proposta do Professor Doutor Edimilson Ribamar, utilizando como referencial o livro “Programming Language Design Concepts”. Neste tópico é realizada a exposição dos conceitos vistos em sala de aula, e como estes foram utilizados no desenvolvimento da aplicação.

* 1. **VALORES E EXPRESSÕES**

Os valores na linguagem Java tem um tipo específico definido explicitamente. Esses tipos variam entre tipos booleanos, inteiros, reais, arrays e objetos. A declaração explícita em Java é feita da seguinte maneira:

|  |
| --- |
| ClasseTeste tclasse = **new** ClasseTeste();  **int** num = 0;  String paralvra;  bolean flag = **false**;  **float** real = 0; |

Em Java também podemos declarar variáveis imutáveis ou seja constantes:

|  |
| --- |
| **final** **float** PI = 3.1416F; **final** String NOME\_PAGINA = "home"; |

Expressões são combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenar strings, comparar valores, realizar operações lógicas e manipular objetos. Sem expressão, uma linguagem de programação seria praticamente inútil. O resultado da avaliação de uma expressão é gerar um valor compatível com os outros tipos de dados que foram operados.

Um exemplo de expressão suportada por JAVA é a construção, segue abaixo um exemplo:

|  |
| --- |
| **class** **Date** {  **public** **int** m, d;  **public** **Date** (**int** m, **int** d) {  **this**.m = m; **this**.d = d;  }  ... } *//Construções de objeto:* Date today = **new** Date(12, 25); Date tomorrow = **new** Date(today.m, today.d+1); |

* 1. **TIPOS COMPOSTO E RECURSIVO**

Tipos compostos são aqueles que podem ser criados a partir de tipos mais simples, podemos exemplificar em Java o conceito de Objeto como sendo um tipo composto, já que este possui tipos mais simples em sua composição, permitindo operações diversas de modificação dos dados. É importante ressaltar que Java trata strings como objetos da classe String, tipo utilizado também na aplicação.

|  |
| --- |
|  |

|  |
| --- |
| *// exemplo de objeto* **public** **class** **WordSet** {    **private** ArrayList<String> words;    **public** **WordSet**(){  *// Construct an empty set of words*  words = **new** ArrayList<>();  }    **public** **void** **add**(String wd){  *// Make wd a member of this set of words.*  **this**.words.add(wd);  }  **public** String **getWord**(**int** index){  **return** words.get(index);  }  **public** **boolean** **contains**(String wd){  *// Return true if and only if wd is a member of this set of words.*  **return** words.contains(wd);   } } |

O array também é exemplo de tipo composto, porém não houve necessidade de aplicação do mesmo no projeto.

O tipo recursivo é um tipo de dado para valores que podem conter outros valores do mesmo tipo. No projeto foi utilizado o ArrayList, representando o tipo recursivo, o mesmo cria uma lista com o objeto escolhido, disponibilizando funções de manipulação de listas. Sua declaração pode ser observada no exemplo dado acima.

* 1. **SISTEMA DE TIPOS**

Estática

A definição básica da tipagem estática que uma linguagem de programação pode ter como característica é que há uma verificação dos tipos usados em dados e variáveis para garantir que sempre está sendo usado um tipo que é esperado em todas as situações. Esta verificação é feita no código fonte pelo processo de compilação. Esta análise ajuda na chamada segurança de tipos na utilização dos dados pelo programa permitindo que o programador se preocupe menos com esta questão. O compilador fornece garantias que alguns problemas não poderão ocorrer após o programa passar por esta verificação, ou seja, erros são detectados logo, antes do programa ser efetivamente executado. Uma variável não pode mudar seu tipo, no entanto a tipagem estática pode causar uma falsa sensação de segurança, só uma parte dos erros podem ser descobertos antecipadamente.

Dinâmica

Na tipagem dinâmica esta verificação também ocorre mas ela é feita em cima do dado em si, já que as variáveis podem conter qualquer tipo de dado. Claro que em determinado momento uma variável só pode conter um tipo de dado e isto é verificado. Mas a principal diferença é que a esta verificação é feita em tempo de execução. Isto é feito através de uma infraestrutura auxiliar (uma máquina virtual ou uma biblioteca normalmente chamada de runtime). É comum o programador ter que fazer suas próprias verificações no programa ou em códigos externos de teste para garantir que todos os tipos estão corretos nos momentos certos. O programador tem que se preocupar mais com os tipos embora em soluções simples pode parecer que a preocupação não seja necessária. O que é dinâmico efetivamente é o tipo da variável. Entender e documentar os tipos ainda são necessários.

Java é uma linguagem estaticamente tipada ou seja, todas as variáveis e expressões têm tipos fixos, tanto estabelecida pelo programador ou inferida pelo compilador. Uma vez que uma variável foi declarada com um tipo em Java ela será até o seu fim do mesmo tipo. Observe também que operações entre tipos diferentes ocasionam erro:

|  |
| --- |
| **int** num1 = 10; String num2 = "5"; **int** num3 = num1 \* num2; *// erro* |

Pode haver exceções onde, em determinadas ocasiões e linguagens, a soma de uma variável do tipo texto(string) com uma variável numérica não ocasiona um erro, mas sim a concatenação dos valores, como no Java, mas isso não significa que a linguagem é tolerante em alguns casos ou que é fracamente tipada, mas sim que aconteceu uma conversão implícita do tipo numérico para string, conversão esta que pode ser feita explicitamente caso o programador deseje.

Existem vantagens e desvantagens sobre as tipagens: a tipagem estática é mais eficiente, pois a dinâmica requer verificação de tipo em tempo de execução. A estática é mais segura ou seja, o compilador pode garantir que o programa objeto não contém erros de tipo, enquanto a tipagem dinâmica não prover tal segurança. Uma vantagem da dinâmica é a sua flexibilidade, pois pode ser necessário em algumas aplicações onde os tipos de dados não são conhecidos previamente.

* 1. **VARIÁVEIS E ARMAZENAMENTO**

Uma variável é um recurso das linguagens de programação utilizado para armazenar valores em memória. Desta forma, sempre que preciso acessar este valor, o mesmo é referenciado através dessa variável. Em Java, podemos declarar variáveis, que podem ter seu valor modificado a qualquer momento, e constantes, estas por sua vez recebem um valor e não pode ser alterada. As variáveis também se diferenciam quanto a simples e compostas, respectivamente ocupando uma célula para armazenamento, ou ocupando um grupo de células.

Declaramos uma variável em Java informando o tipo de dados que ela poderá receber e seu nome. Essa variável pode estar dentro de um método utilizando somente a declaração dita anteriormente, porém, sendo esta uma variável de classe antes da declaração do tipo especificamos o modificador de acesso, além da necessidade de inicializá-la através de construtor.

* 1. **VÍNCULOS E ESCOPO**

Os comandos e expressões definidos no corpo da codificação contém identificadores, o vínculo é definido como a relação entre este identificador e o procedimento, variável ou valor. Já o ambiente seria o conjunto dos vínculos.

|  |
| --- |
| *// trecho de código representando procedimento , seu nome é usado como identificador do procedimento, relacionado com o objeto*   **public** **void** **printDict**(){  **for** (String dict1 : dict) {  System.out.println(dict1);  } |

|  |
| --- |
|  |

Escopo tem como definição a relação onde a declaração e vínculos se limitam. Em Java, podemos exemplificar o escopo com a declaração de uma variável, sendo esta válida dentro das chaves onde é declarada, se á variável é declarada “dentro” de um método, ela só é válida e alterável dentro de tal método, sendo os limites do método seu escopo. Porém se a variável for declarada fora de qualquer método, sendo então variável de classe (atributo), seu escopo se torna a classe inteira, podendo alterar e utilizar seu valor em qualquer método desta classe.

Podemos definir também Java como sendo *estaticamente escopada*, ou seja, o corpo de um método é executado no ambiente onde este foi definido.

* 1. **ABSTRAÇÃO PROCEDURAL**

Em programação o significado de abstração é distinção entre o que a unidade de programa faz e como ela faz.

Um procedimento função abstrae sobre uma expressão. Um procedimento função tem um corpo que é uma expressão, e uma chamada de função é uma expressão que fornecerá um valor avaliando o corpo do procedimento função. Um procedimento próprio abstrai sobre um comando. Um procedimento próprio tem um corpo que é um comando, e uma chamada de procedimento é um comando que atualizará variáveis executando o corpo do procedimento próprio.

As linguagens orientada a objetos tem como a única forma de procedimento o método e Java se enquadra nesse quesito.

Parâmetros e argumentos, argumentos geralmente são valores ou entidades que são passados para um procedimento, parâmetro é uma expressão que gera um argumento. Existem três possíveis mecanismos de passagem de parâmetros por cópia: passagem de parâmetros por cópia na entrada, passagem de parâmetros por cópia na saída, passagem de parâmetros por cópia na entrada e saída, e também existem três mecanismos de passagem por referência: parâmetros constantes, parâmetros variáveis, parâmetros procedurais.

Java suporta mecanismo de passagem de parâmetros por cópia na entrada para primitivos e mecanismos de parâmetro por referência para tipos objetos: Parâmetros do tipo int, float, etc. são passados por cópia. Parâmetros do tipo T[], String, List, etc. são passados por referência.

* 1. **ABSTRAÇÃO DE DADOS**

Java suporta abstração de dados ou seja pacotes, encapsulamento, classes e herança. Um pacote é um denominado grupo de componentes declarados para um propósito comum, esses componentes podem ser tipos, constantes, variáveis e etc. ou seja um pacote é uma coleção de classes relacionadas que provê acesso protegido e gerenciamento de espaço de nomes. Uma das grandes motivações de se usar pacotes é para que outros programadores possam determinar facilmente que estes tipos são relacionados. Para criar um pacote basta colocar todos os tipos(classes, interfaces, etc.) dentro dele e a primeira linha de cada arquivo fonte deve conter o comando package seguido do nome do pacote. Segue abaixo um exemplo:

|  |
| --- |
| **package** corretor; **public** **class** **Dictionary extends** **WordSet** { . . .} |

Encapsulamento significa fazer alguns componentes de uma unidade de programa privados, sejam eles pacotes ou classes. Existem três níveis de privacidade possíveis privado, protegido e público. Um componente é privado se é visível somente dentro da unidade de programa. Um componente é protegido se é visível somente dentro da unidade de programa e certas unidades de programas proximamente relacionadas. Um componente é público se é visível para código de aplicação que está fora da unidade de programa.

Herança é um mecanismo para derivar novas classes a partir de classes existentes, a classe derivada herda a representação de dados e operações de sua classe base. Pode-se adicionar novas operações, estender a representação dos dados ou redefinir a implementação de operações existentes. Herança simples permite cada classe ter no máximo uma superclasse. Herança múltipla permite que cada classe tenha qualquer número de superclasses. Java so suporta herança simples. Exemplo de herança em java:

|  |
| --- |
| **public** **class** **Dictionary** **extends** **WordSet**{    **private** ArrayList<String> dict;  **private** String line;  **private** FileWriter arq;   **private** PrintWriter saveArq;   **public** **Dictionary**(){  *//construct an empty dictionary ...*   dict = **new** ArrayList<>();  } ... } |

* 1. **ABSTRAÇÃO GENÉRICA**

Uma unidade genérica é uma unidade de programa que é parametrizada com respeito a entidades em que ela depende. Instanciação de uma unidade genérica gera uma unidade de programa comum, em que cada um dos parâmetros formais da unidade genérica é substituído por um argumento. Até 2004 Java não suportava qualquer forma de abstração genérica, porém agora suporta classes genéricas, que na verdade são classes parametrizadas com respeito a outras classes. Se uma classe genérica Java assume que um parâmetro de classe é equipado com operações particulares, o parâmetro de classe deve ser especificada para implementar uma interface adequada. Exemplo de classe genérica em Java:

|  |
| --- |
| *//Esta classe genérica em java encapsula listas homogêneas. A classe é parametrizada / /com respeito a classe dos elementos da lista.* **class** **List** <**Element**> { . . . **public** **void** **append** (Element e) { *// Add element e to the end of this list.* . . . } . . . *// other methods* } |

* 1. **EXCEÇÕES**

Para definir exceções, vamos introduzir alguns conceitos importantes. Inicialmente temos o conceito de sequenciadores, estes têm como função transferir o controle para parte do programa. Com isso podemos definir salto e escape, o primeiro é um sequenciador específico, transferindo o controle para uma localização específica do código, e o segundo é um sequenciador que finaliza a execução de um determinado bloco de comandos ou procedimentos.

|  |
| --- |
| *// switch case representando o uso de escape*  **public** **static** String **consultUser**(String currentWord, Dictionary mainDict, Dictionary ignored) **throws** IOException{ ....  chooseUser = userInput.nextInt();    **switch** (chooseUser) {  **case** 0:  mainDict.addDict(currentWord);  mainDict.save("main-dict.txt");  *// mainDict.printDict();*  **break**;  **case** 1:  ignored.addDict(currentWord);  ignored.save("ignored.txt");  **break**;  .... } |

Esses sistemas de escolha já são considerados na produção do código sendo parte do fluxo de execução, deste modo qualquer umas das escolhas feitas, gera um execução de um determinado bloco de código, e em seguida a execução do programa prossegue sem problemas. Porém, existem situações anormais, que podem causar interrupções na execução do programa. Na maioria das linguagens, erros deste tipo não são solucionados em tempo de execução, encerrando o fluxo do programa quando esse tipo de situação ocorre, em contrapartida existem em algumas linguagens o handler, em que quando uma situação anormal é detectada em alguma unidade de programa no baixo nível, este trecho de código permite ao programa se recuperar-se da situação.

Temos então a exceção, representando estas situações anormais, as identificando e verificando.

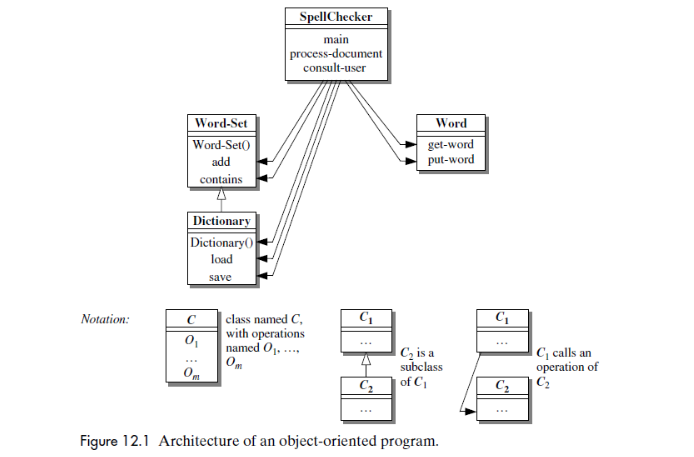
Em Java, podemos utilizar a classe Exception para tratamento dessas situações anormais, toda exceção é um objeto de uma subclasse de Exception, onde cada subclasse representa uma situação de exceção diferente e a partir de cláusulas definidas pelo desenvolvedor, realiza soluções ou desvios, caso a exceção seja identificada, sem a necessidade de interromper a execução .

|  |
| --- |
| *// exemplo de método que usa exceções para leitura*  **public** **void** **load**(String fileName){  *// Load this dictionary from filename*  **try**{  BufferedReader lerArqPerg = **new** BufferedReader(**new** FileReader(fileName));  line = lerArqPerg.readLine();   **while** (line != **null**) {  dict.add(line);  line = lerArqPerg.readLine();  }  lerArqPerg.close();  }**catch**(IOException e){  System.err.printf("Erro na abertura do arquivo: %s.\n", e.getMessage());  }  System.out.println("Lido e salvo com sucesso"); ...  } |

O bloco try tenta processar o código que está dentro, sendo que se ocorrer uma exceção, a execução do código pula para a primeira captura do erro no bloco catch. O uso do try serve para indicar que o código está tentando realizar algo arriscado no sistema.  
 O bloco catch trata a exceção lançada. Caso a exceção não seja esperada, a execução do código pula para o próximo catch, se existir. Portanto, se nenhum do bloco catch conseguir capturar a exceção, dependendo o tipo que for, é causada a interrupção ao sistema, lançando a exceção do erro.

**3. CONSIDERAÇÕES FINAIS**

Foi realizada a produção do projeto seguindo os conceitos apresentados em sala de aula, e expostos neste documento. A utilização da LP Java, juntamente com o paradigma OO possibilitaram uma facilidade na construção e entendimento do código da aplicação, a partir da abstração do problema como objeto suportada por tais.

Para solução do problema proposto foi preciso armazenar as informações, tanto o texto do usuário como o dicionário, em estruturas de dados, conseguindo assim manipular da maneira necessária. Ainda seguindo com a solução do problema foi necessário sempre checar se a palavra recebida existe no dicionário, caso existisse, não era feita a verificação da correção, somente era escrita no arquivo de saída e quando a palavra não estivesse no dicionário era proposto a verificação da correção, listando três possíveis opções: adicionar ao arquivo de dicionário, adicionar ao arquivo de ignorados e por último era reescrever palavra de forma correta, para então a palavra ser escrita no arquivo de saída. É possível verificar o fluxo da execução do código em questão , analisando o diagrama a seguir: 

Foram exploradas todas as possíveis funcionalidades disponibilizadas pela linguagem, de acordo com sua necessidade de aplicação, sempre seguindo os conceitos de OO. O código final, apresentou uma solução para o problema proposto, visto que foi implementado a solução baseada no problema base. No decorrer da implementação foram detectadas possíveis melhoras na idealização de soluções, como propor ao usuário sugestões de correção para a suposta palavra errada, usando um sistema de verificações de substring, relacionando String “errada” com as mais próximas possíveis a ela, objetivamos trazer mais robustez ao programa, a partir desta e outras melhorias, levando em conta a facilidade fornecida pela linguagem, e objetivando levar facilidade de uso e desempenho para o usuário.

**REFERÊNCIAS**

JAVA : PRINCIPAIS CARACTERÍSTICAS. GRUPO. Disponível em: <<http://lpjava.blogspot.com.br/2007/11/principais-caractersticas.html>>.

Acesso em: 15 ago. 2017.

LINGUAGEM DE PROGRAMAÇÃO JAVA

<<https://pt.wikiversity.org/wiki/Introdu%C3%A7%C3%A3o_%C3%A0s_Linguagens_de_Programa%C3%A7%C3%A3o/Java>>

Acesso em 14 de ago. 2017

CARACTERÍSTICAS JAVA

<http://www2.unoeste.br/~aglae/ling\_pro/java\_caracteristicas.htm>

Acesso em: 15 de ago. 2017

David & Findlay, Programming Language Design Concepts,

John Wiley & Sons, Ltd, 2004