Министерство науки и высшего образования РФ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Отчет по лабораторной работе №2

**Использование средств автоматической генерации документации и рефакторинга программного кода**

дисциплина «Технология разработки программного обеспечения»

|  |  |
| --- | --- |
| Выполнил: студент группы ИВТм-1301 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Савин Д.А. / |
|  |  |
| Проверил: доцент кафедры ЭВМ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Частиков А.В. / |

Киров 2022

**Цель**: Целью работы является знакомство с встроенными в среду разработки Eclipse средствами для автоматической генерации документации и рефакторинга кода.

**Задачи**:

1. Провести ряд преобразований программного кода, полученного в ходе выполнения предыдущей работы, с использованием встроенных средств рефакторинга.
2. Сопроводить код комментариями с использованием JavaDoc.
3. Сгенерировать документацию к разработанным классам.

Класс SmallInteger:

|  |
| --- |
| **package** SmallInteger\_Class;  **import** **java.util.Arrays**;  /\*\*  \* Class for working with integers. The class contains public methods for addition, subtraction,  \* multiplication and division of integers not exceeding in absolute  \* value 10^4.  \* @since 28.11.2022  \*/  **public** **class** **SmallInteger** **implements** Comparable<SmallInteger>  {  **private** **boolean** POSITIVE = **true**;  **private** **byte** [] oldValue;  **byte** [] numbers;  **final** **static** SmallInteger ZERO;  **static** {  **try** {  ZERO = **new** SmallInteger("0");  } **catch** (Exception e) {  **throw** **new** **RuntimeException**(e);  }  }  **private** **SmallInteger**(**byte**[] digits)  {  numbers = digits;  }  /\*\*  \* Translates the decimal String representation of a SmallInteger into a SmallInteger.  \* @param val decimal String representation of SmallInteger  \* @throws Exception {@code val} is not a valid representation of a SmallInteger.  \*/  **public** **SmallInteger**(String val) **throws** Exception {  **if** (Integer.parseInt(val) <= Math.pow(**10**, **4**)) {  **this**.numbers = **new** **byte**[val.length()];  **for** (**int** i = **0**; i < val.length(); i++) {  numbers[i] = (**byte**) (val.charAt(i) - **48**);  }  oldValue = numbers.clone();  } **else** {  **throw** **new** **Exception**("BIG NUMBER");  }  }  /\*\*  \* Returns the decimal String representation of this SmallInteger.  \* @return decimal String representation of this SmallInteger  \*/  **public** String **toString**() {  StringBuilder sb = **new** StringBuilder();  **for** (**byte** number : **this**.numbers) {  sb.append(number);  }  **return** sb.toString();  }  /\*\*  \* Returns the number of digits in a number.  \* @return returns the number of digits in a number.  \*/  **public** **int** **length**() {  **return** **this**.numbers.length;  }  /\*\*  \* Compares this SmallInteger with the specified SmallInteger.  \* @param val SmallInteger to which this SmallInteger is to be compared.  \* @return -1, 0 or 1 as this BigInteger is numerically less than, equal  \* to, or greater than {@code val}.  \*/  **@Override**  **public** **int** **compareTo**(SmallInteger val) {  **if**(val.length()>**this**.length()) **return** -**1**;  **if**(val.length()<**this**.length()) **return** **1**;  **if**(val.length()==**this**.length()) {  **int** idx = **0**;  **while**(idx<=**this**.length()-**1**) {  **if**(val.numbers[idx]>**this**.numbers[idx]) **return** -**1**;  **if**(val.numbers[idx]<**this**.numbers[idx]) **return** **1**;  idx++;  }  }  **return** **0**;  }  /\*\*  \* Returns a SmallInteger whose value is {@code (this - val)}.  \* @param val value to be subtracted from this SmallInteger  \* @return {@code this - val}  \* @throws Exception {@code result} is not a valid representation of a SmallInteger.  \*/  **public** SmallInteger **subtract**(SmallInteger val) **throws** Exception {  **if** (val.compareTo(**new** SmallInteger("0")) == **0**)  **return** **new** **SmallInteger**(**this**.numbers);  **if** (**this**.compareTo(val) < **0**) {  **byte**[] temps = Arrays.copyOf(val.numbers, val.length());  val.\_subtract(**this**);  **this**.numbers = val.numbers;  val.numbers = temps;  POSITIVE = **false**;  }  **else** {  **if** (**this**.compareTo(val) == **0**) {  **this**.numbers = ZERO.numbers;  } **else** {  **this**.\_subtract(val);  }  }  SmallInteger result = **new** SmallInteger(**this**.numbers);  numbers = oldValue.clone();  POSITIVE = **true**;  **return** result;  }  /\*\*  \* Subtracts the contents of the second int arrays (little) from the  \* first (big). The first int array (big) must represent a larger number  \* than the second. This method allocates the space necessary to hold the  \* answer.  \*/  **private** **void** **\_subtract**(SmallInteger val) {  **if**(val.length()>**this**.length()) {  **this**.shiftLeft((val.length()-**this**.length()));  }  **for**(**int** i = **0**; i < val.length(); i++) {  **byte** tempVal = (**byte**) (numbers[**this**.length() - i - **1**] - val.numbers[val.length() - i - **1**]);  **if** (numbers[**this**.length() - i - **1**] - val.numbers[val.length() - i - **1**] < **0**) {  **if** (**this**.length() - i - **2** < **0**) {  numbers[**this**.length() - i - **1**] = (**byte**) (tempVal + (**byte**) **1**);  **return**;  }  **if** (numbers[**this**.length() - i - **1**] < -**1**){  tempVal += **1**;  }  numbers[**this**.length() - i - **1**] = (**byte**) (**10**+ (tempVal));  numbers[**this**.length() - i - **2**] += (**byte**)((tempVal / **11**) -**1**);  } **else** {  numbers[**this**.length() - i - **1**] = (tempVal);  }  }  **if** (findBad(**this**))  {  **int** SIZE;  **if** (**this**.numbers[**0**] >= **1** && val.numbers.length <= **2**)  {  SIZE = **this**.numbers.length;  } **else**  {  SIZE = (**this**.numbers.length - **1**);  }  **if** (**this**.numbers[**1**] == -**1**) **this**.numbers[**0**] -= **1**;  **if** (SIZE == **5**) SIZE--;  **byte**[] tmpVal = **new** **byte**[SIZE];  **boolean** m = **false**;  **for** (**int** i = tmpVal.length - **1**; i >= **0**; i--)  {  tmpVal[i] = **9**;  **if** (**this**.numbers.length == tmpVal.length)  {  **if** (m)  {  **if** (i!=**0** && numbers[i]!=**0**)  tmpVal[i] = (**byte**) (numbers[i]+numbers[i+**1**]);  **else**  **if** (numbers[i] != **0**)  tmpVal[i] = (numbers[i]);  **else**  {  **if** (i==**0**)  {  tmpVal[i] = **0**;  **break**;  }  tmpVal[i] = **9**;  tmpVal[i-**1**] = (**byte**) (numbers[i-**1**]-**1**);  **break**;  }  m = **false**;  **continue**;  }  **if** (i!= **0** && **this**.numbers[i] == **0** && **this**.numbers[i-**1**] == -**1**)  tmpVal[i] = **this**.numbers[i];  **else**  **if** (**this**.numbers[i] == **0** && i == tmpVal.length - **1**)  tmpVal[i] = **this**.numbers[i];  **else**  **if** (**this**.numbers[i] > **0**)  tmpVal[i] = **this**.numbers[i];  **else** m=**true**;  }  **else**  {  **if** (**this**.numbers[i + **1**] == **0** && (i != **0** || i + **1** == **this**.numbers.length - **1**) && (**this**.numbers[i] != **0** || i-**1** != **0**))  tmpVal[i] = **this**.numbers[i + **1**];  **else** **if** (**this**.numbers[i + **1**] > **0**)  tmpVal[i] = **this**.numbers[i + **1**];  **if** (i + **2** < **this**.numbers.length && **this**.numbers[i + **1**] == **0** && **this**.numbers[i] < **0**)  {  tmpVal[i] = **0**;  }  }  }  **this**.numbers = tmpVal;  }  **this**.autoShrink();  }  **private** **boolean** **findBad**(SmallInteger val){  **for** (**int** i=**0**; i<val.numbers.length; i++)  {  **if** (val.numbers[i] < **0**){  **return** **true**;  }  }  **return** **false**;  }  /\*\*  \* Returns a SmallInteger whose value is {@code (this + val)}.  \* @param val value to be added to this SmallInteger.  \* @return {@code this + val}  \*/  **public** SmallInteger **add**(SmallInteger val)  {  **if**(val.length()>**this**.length()) {  **this**.shiftLeft((val.length()-**this**.length()));  }  **int** idx = **1**;  **if**(POSITIVE)  **while**(idx <= val.length() ) {  **this**.numbers[**this**.length()-idx] += val.numbers[val.length()-idx];  idx++;  }  **else** **\_subtract**(val);  carry(**this**.length()-**1**, **this**.length()-val.length()-**2**);  SmallInteger result = **new** SmallInteger(**this**.numbers);  numbers = oldValue.clone();  **return** result;  }  /\*\*  \* Returns a SmallInteger whose value is {@code (this \* val)}.  \* @param val value to be multiplied by this SmallInteger.  \* @return {@code this \* val}  \*/  **public** SmallInteger **multiply**(SmallInteger val)  {  **byte**[] temp = **new** **byte**[val.length()+**this**.length()+**1**];  **if**(**this**.length()<val.length()) **this**.shiftLeft(val.length()-**this**.length());  **int** move = **0**;  **for**(**int** i = **this**.length()-**1**; i >= **0**; i--) {  **for**(**int** s = val.length()-**1**; s >= **0**; s--) {  **if**(temp.length-**1**-move <= **0**) **break**;  **if**(((**int**)temp[temp.length-**1**-move] + (**int**)val.numbers[s]\*(**int**)**this**.numbers[i]) > **100** ) carry(temp,temp.length-**1**-move,**0**);  temp[temp.length-**1**-move] += val.numbers[s]\***this**.numbers[i];  move++;  }  move = **this**.length()-i;  }  carry(temp,temp.length-**1**,**0**);  **this**.numbers = temp;  **this**.autoShrink();  SmallInteger result = **new** SmallInteger(**this**.numbers);  numbers = oldValue.clone();  **return** result;  }  **public** Integer **toInteger**() {  **int** temp = **0**;  **if**(length()>=**10** || length()==**10** && numbers[**0**]>=**2**) **return** **null**;  **for**(**int** i = **0**; i < length(); i++) {  **int** placeVal = (**int**)Math.pow(**10**, length()-**1**-i)\*numbers[i];  temp += placeVal;  }  **return** temp;  }  /\*\*  \* Returns a SmallInteger whose value is {@code (this % val)}.  \* @param val value by which this SmallInteger is to be divided, and the  \* remainder computed.  \* @return {@code this % val}  \*/  **public** **int** **remainderOfDivision**(SmallInteger val)  {  **int** num1 = **this**.toInteger();  **int** num2 = val.toInteger();  **while** (num1 - num2 >= **0**){  num1 -= num2;  }  **return** num1;  }  /\*\*  \* Returns a BigInteger whose value is {@code (this / val)}.  \* @param val value by which this SmallInteger is to be divided.  \* @return {@code this / val}  \* @throws Exception {@code new val} is not a valid representation of a SmallInteger  \*/  **public** SmallInteger **divide**(SmallInteger val) **throws** Exception  {  **int** num1 = **this**.toInteger();  **int** num2 = val.toInteger();  **int** q;  **for** (q = **0**; num1 - num2 >= **0**; q++)  num1 -= num2;  **return** **new** **SmallInteger**(String.valueOf(q));  }  /\*\*  \* removing unnecessary bits at the beginning.  \*/  **private** **void** **autoShrink**() {  **int** i = **0**;  **while**(i < **this**.length()-**1** && numbers[i]==**0**) {  i++;  }  **byte**[] newArray = **new** **byte**[numbers.length-i];  System.arraycopy(numbers, i, newArray, **0**, numbers.length-i);  **this**.numbers=newArray;  }  **private** **void** **shiftLeft**(**int** val) {  **byte**[] temp = **new** **byte**[numbers.length+val];  System.arraycopy(numbers, **0**, temp, val, numbers.length);  numbers = temp;  }  **public** **void** **carry**(**int** idx, **int** end) {  **if**(idx == **0** || idx == end) **return**;  **if**(numbers[idx] < **10** && numbers[idx] >= **0**){  **while**(numbers[idx] < **10** && numbers[idx] >= **0**) {  idx--;  **if**(idx < **0**) **return**;  **if**(idx-**1** <= **0**) **return**;  }  }  **if**(numbers[idx] >= **10**){  numbers[idx-**1**] += (**byte**) (numbers[idx] / **10**);  numbers[idx] = (**byte**) (numbers[idx] % **10**);  }**else** **if**(numbers[idx] < **0**) {  numbers[idx]++;  }  carry(idx-**1**, end);  }  **private** **void** **carry**(**byte**[] array, **int** idx, **int** end) {  **if**(idx == **0** || idx == end) **return**;  **if**(array[idx] < **10** ) {  **while**(array[idx] < **10**) {  idx--;  **if**(idx < **0**) **return**;  **if**(idx-**1** <= **0**) **return**;  }  }  **if**(array[idx] >= **10**){  array[idx-**1**] += (**byte**) (array[idx] / **10**);  array[idx] = (**byte**) (array[idx] % **10**);  }  carry(array, idx-**1**, end);  }  } |

Вывод:

Был проведен рефакторинг кода с помощью встроенных средств среды разработки, сопроводили код комментариями и сгенерировали документацию к разработанным методам.