SWP4 ÜBUNG

Beispiel 1: Observer Blocks

1. Lösungsidee

Das Observer-Pattern wurde hier intensiv eingesetzt. Im Grunde gibt es drei Grund-Typen an Blöcken: Emit-Block, Intermediate-Block und Terminal-Block. Davon abgeleitet gibt es Sub-Typen, die ihre eigenen Funktionen haben.

Im ersten Beispiel gibt es nur drei Ausprägungen: FileReaderBlock (von Emit-Block), SumBlock (von Intermediate-Block) und ConsoleBlock (von Terminal-Block).

Der FileReaderBlock liest eine Datei zeilenweise ein und sendet die Zeilen an seinen Observer (hier SumBlock). SumBlock summiert pro Zeile auf und sendet das Endergebnis an den ConsoleBlock, der das Ergebnis auf die Konsole hinaus schreibt.

Die zweite Variante davon ist komplizierter. Enthalten sind: FileReaderBlock (EmitBlock), BufferBlock (IntermediateBlock), MedianBlock (IntermediateBlock), DifferentialBlock (IntermediateBlock), MinBlock (IntermediateBlock), MaxBlock (IntermediateBlock), AvgBlock (IntermediateBlock), ConsoleBlock (EmitBlock) und FileWriterBlock (EmitBlock).

Zuerst liest der FileReaderBlock eine Datei (sample_data.csv) ein und schickt zeilenweise die Ergebnisse an seinen Observer BufferBlock. BufferBlock kann dazu konfiguriert werden, eine gewisse Anzahl an Zeilen zu speichern, bevor er sie an seinen Observer, MedianBlock, schickt. Bei diesem kann man eine Fensterbreite konfigurieren, wie er den Median berechnet. Hat er alle Mediane berechnet, schickt er die Ergebnisse an seinen Observer DifferentialBlock. Dieser berechnet die Differenz auf Basis des Differentialquotientens und schickt das Ergebnis an die Observer MinBlock, MaxBlock, AvgBlock und FileWriterBlock. Während die ersten drei das Minimum, Maximum und den Durchschnitt berechnen, schreibt der letzte Block die Ergebnisse der Differenz in eine eigene .csv-Datei hinaus (den Filenamen kann man auch angeben). Letzten Endes schicken kann Min, Max und Avg ihre Ergebnisse an den finalen Observer ConsoleBlock, der die Ergebnisse auf die Konsole hinaus wirft.

Alle IntermediateBlock-Typen sind sowohl Observer als auch Observable. EmitBlocks sind nur Observable und TerminalBlocks nur Observer.

2. Testfälle

Anmerkung: BlockTest enthält die Testfälle für den ersten Fall. FancyBlockTest enthält die für den zweiten Fall!

Die Anwendung nimmt keine Strings an!

Test01:

Liest einen Filenamen über die Konsole ein.

Das Programm summiert die einzelnen Zeilen auf und sendet das Ergebnis an die Konsole.

```
"C:\Program Files\Java\jdk1.8.0_172\bin\java.exe" ...
IntegerTest.txt
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 74.0
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 505.0
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 420.0
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 0.0
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 238.0
```

Ein falscher Filename wird angegeben. Daraufhin macht das Programm nichts.

```
"C:\Program Files\Java\jdk1.8.0_172\bin\java.exe" ...
IDonExist.txt

Process finished with exit code 0
```

Test02:

Hier wird der Filename über die Methode create_stream() übergeben. Ist dasselbe Ergebnis wie bei 01, da es Integer.txt war.

Test03:

Liest über die Methode ein. IntegerTest2.txt beinhaltet negative Werte.

```
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: -25.0
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 83.0
```

Test04:

Beinhaltet Double-Werte.

```
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: 2.3200000000000003
Received value from Beispiel_01.Blocks.Implementations.SumBlock@4554617c: -39.995000000000005
```

Test05:

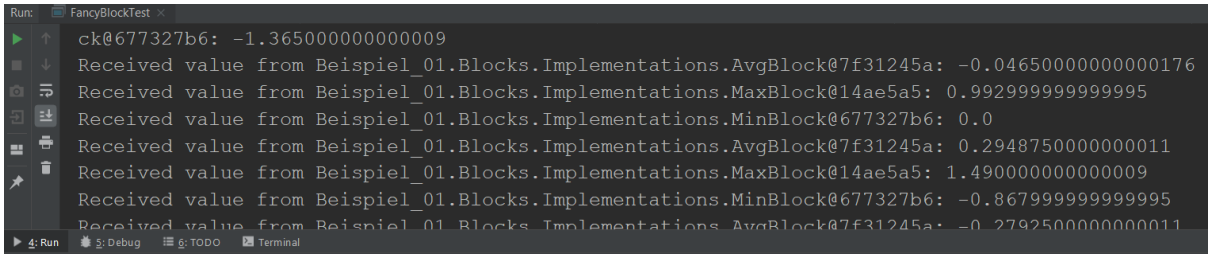
Übergibt eine Datei StringTest.txt, die nur Strings enthält. Funktioniert natürlich nicht. 😊

FancyBlockTest:

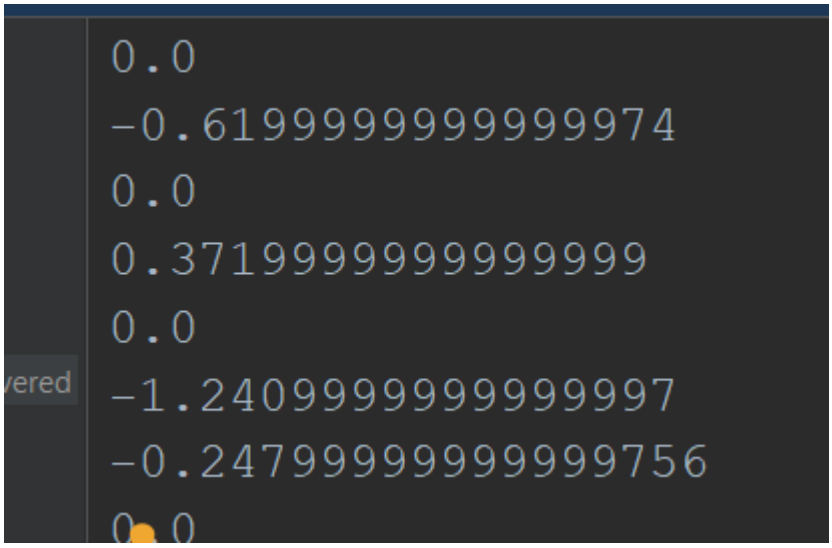
Test01:

Initialisiert alle nötigen Blocks und liest von der Datei sample_data.csv ein. Die Ergebnisse von Min, Max und Avg werden auf die Konsole ausgegeben. Die Differenzen werden in die Datei signal.csv geschrieben.

Hier ein minimaler Auszug von den Ausgaben auf die Konsole:



Auszug aus der Datei signal.csv:



Test02:

Es wird nur der FileReaderBlock und der ConsoleBlock initialisiert. Der FileReaderBlock liest Zeilen ein und wirft sie an den ConsoleBlock weiter.

```
FileReaderBlock@6e0be858: 89.112,3.794,12.42
Received value from Beispiel_01.Blocks.Implementations.FileReaderBlock@6e0be858: 89.36,3.794,12.4202
Received value from Beispiel_01.Blocks.Implementations.FileReaderBlock@6e0be858: 89.608,3.796,12.4204
Received value from Beispiel_01.Blocks.Implementations.FileReaderBlock@6e0be858: 89.112,3.798,12.4206
Received value from Beispiel_01.Blocks.Implementations.FileReaderBlock@6e0be858: 88.367,3.798,12.4208
```

Test03:

Nun kommt der BufferBlock dazwischen. Dieser sammelt eine Anzahl an Zeilen (hier zwei) und gibt die Werte weiter.

Wichtig: Pro Zeile gibt es drei Werte. Bei einem BufferBlock der Größe zwei (wie in diesem Fall) werden also zwei Zeilen gespeichert. 2 x 3 = 6 Werte werden an den ConsoleBlock weiter geworfen.

```
554617c: [385.242,3.794,9.2982, 384.746,3.794,9.2984]
Received value from Beispiel_01.Blocks.Implementations.BufferBlock@4554617c: [384.125,3.79,9.2988, 385.615,3.792,9.299
Received value from Beispiel_01.Blocks.Implementations.BufferBlock@4554617c: [384.746,3.792,9.2994, 385.615,3.792,9.29
Received value from Beispiel_01.Blocks.Implementations.BufferBlock@4554617c: [383.629,3.796,9.3, 387.352,3.794,9.3002]
```

Test04:

Hier kommt nun der MedianBlock nach BufferBlock. Dieser bildet mit einer spezifizierten Fenstergröße den Median über die gebufferten Werte. Wichtige Anmerkung: Es macht keinen Sinn eine Fenstergröße zu nehmen, die größer als der Buffer ist, also ein Fenster von 3 auf einen Buffer von 2 zu legen. Es werden nämlich nur immer die ersten Werte der Zeile genommen, da diese die relevanten Daten beinhalten. Also pro Zeile nur ein Wert.

Hier wurde ein Buffer von 6 und eine Fenstergröße von 3 genommen.

```
from Beispiel_01.Blocks.Implementations.MedianBlock@74a14482: [401.268, 401.268, 399.779, 399.53, 399.53, 399.53]
from Beispiel_01.Blocks.Implementations.MedianBlock@74a14482: [398.537, 398.537, 399.282, 399.53, 399.53, 399.53]
from Beispiel_01.Blocks.Implementations.MedianBlock@74a14482: [399.034, 399.034, 398.289, 398.289, 398.289, 398.289]
from Beispiel_01.Blocks.Implementations.MedianBlock@74a14482: [400.399, 400.399, 400.523, 400.399, 400.399, 400.399]
```

Test05:

Jetzt kommt der DifferentialBlock dazu. Dieser nimmt den Wert an der Stelle Index + 1 und zieht davon den Wert an der Stelle Index ab. Da die Fenstergröße beim Median nur 3 war und der Buffer nur 6, kommen teilweise für mehrere Werte pro Medianberechnung die gleichen Werte, daher 0-Werte!

```
iel_01.Blocks.Implementations.DifferentialBlock@1540e19d: [0.0, 0.1239999999999668, 0.0, 0.0, 0.0]
iel_01.Blocks.Implementations.DifferentialBlock@1540e19d: [0.0, -0.37199999999995725, 1.7379999999999995, 0.0, 0.0]
iel_01.Blocks.Implementations.DifferentialBlock@1540e19d: [0.0, -0.1239999999999668, -1.2409999999999854, 0.0, 0.0]
iel_01.Blocks.Implementations.DifferentialBlock@1540e19d: [0.0, -0.8689999999999714, 0.0, 0.0, 0.0]
iel_01.Blocks.Implementations.DifferentialBlock@1540e19d: [0.0, 0.37299999999999045, 0.0, 0.0, 0.0]
```

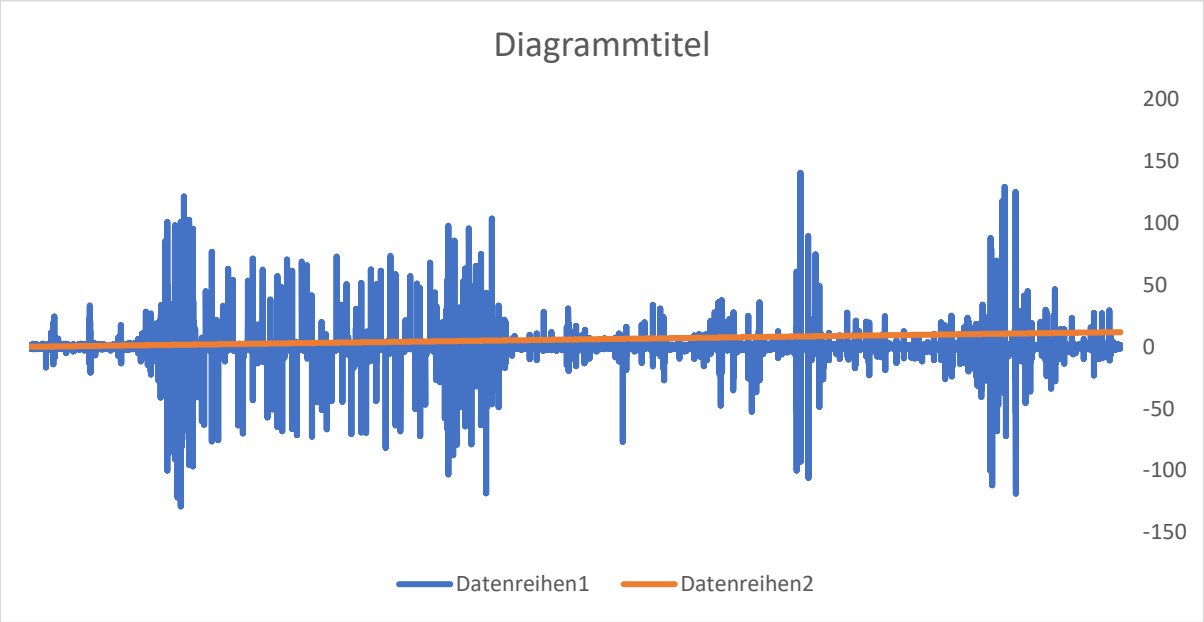
Test06:

Min, Max, Avg und FileWriterBlock wurden in Test01 schon gezeigt. 😊

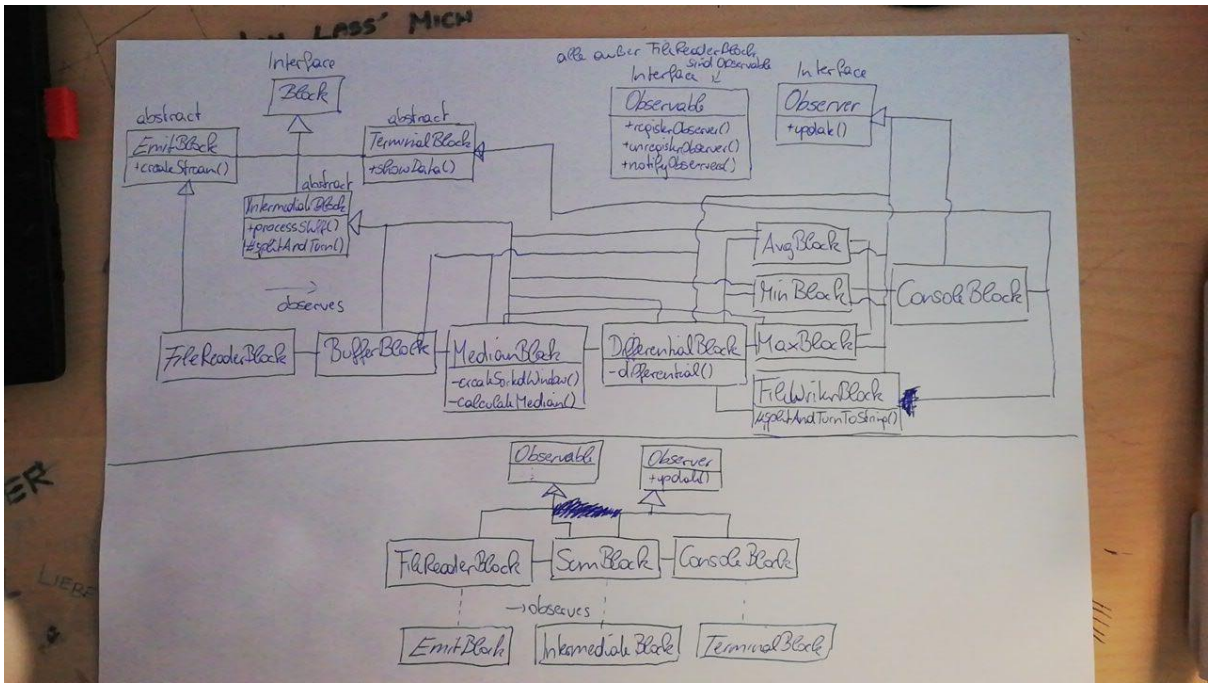
Analyse von signal.csv:

Es ist leider schlecht zu sehen, da ich nicht so recht wusste, wie ich die Daten sonst darstellen soll.

Die orange Linie repräsentiert die Zeit, während die blauen Balken die gemessenen Werte sind.



Klassendiagramm: (Leider etwas durcheinander. Observable wurde nicht mehr verbunden, hat aber eine Verbindung zu allem, außer FileReaderBlock)



Interessante Sequenzdiagramme gibt es nicht, da der Ablauf eigentlich immer gleich ist:

Irgendein Block ruft die Grundfunktion auf (bei EmitBlock z.B. create_stream). Am Ende dieser Funktion wird notifyObservers() aufgerufen.

Bei IntermediateBlock wird in update() dann process_stuff() aufgerufen (diese ruft vielleicht eigene Funktionalitäten auf) und am Ende dann notifyObservers().

Der TerminalBlock wird per update() aufgerufen und öffnet hier die eigene Funktion show_data().

Der Ablauf ist immer gleich!

Der Quellcode wird wieder in eine eigene Datei ausgelagert. 😊

Eine Frage habe ich auch: Gibt es eine bessere Möglichkeit, wie man mit der IOException umgehen kann, die vom FileWriter ausgelöst wird? Mit try-catch wird zwar die Exception gefangen, aber der ganze Prozess wird dadurch gestoppt (es wird ein Wert hinaus geschrieben, dann wird der Fehler geworfen und der catch() Block wird aufgerufen).

Meine Lösung war, dass ich die Exception einfach immer weiter geworfen habe, aber das ist garantiert nicht der passende Ansatz. Gibt es eine bessere Lösung? :/

Beispiel 02

1. Lösungsidee

Bei den FileUtilities gibt es mehrere gewünschte Funktionen.

Die erste ist die Funktion Head, die einem die gewünschte Anzahl an Zeilen einer Textdatei ausgibt. Von der Laufzeit her ist diese hier leichter, da man gleich beim Einlesen mitzählen kann, wie viele Zeilen man braucht. Leerzeilen werden hierbei nicht gezählt!

Die zweite ist die Funktion Tail. Diese gibt einem die gewünschten n-letzten-Zeilen einer Textdatei zurück. Hier ist es etwas aufwändiger, da zuerst die Anzahl der Zeilen der Textdatei ermittelt werden müssen, bevor man weiß, wie viele man davon braucht.

LOC nimmt einfach nur eine Datei und zählt die Zeilen, einzelnen Wörter und Buchstaben. Hierbei werden die Wörter nur von Leerzeichen getrennt, sonst wird es nicht anders gezählt.

TreeSize beginnt bei einem Pfad und arbeitet sich von dort aus rekursiv nach unten weiter. Dank der java.io.File kann man relativ einfach durch das Dateisystem navigieren. Im Grunde wird ein File (eigentlich ein Directory) mit dem Path erstellt, dann wird daraus die Liste der einzelnen Files (und Directories) generiert, durch welche das Programm hindurch geht und summiert.
Wichtige Anmerkung: Da es rekursiv ist, kann es schnell passieren, dass die Funktion länger dauert!
Weitere Anmerkung: Es wird immer der gesamte Pfad bei der Ausgabe angegeben, daher könnte es ein wenig unübersichtlich werden. Tut mir Leid! :/

2. Testfälle

Für Head und Tail:

Test01():

Liest aus einer Textdatei Monika.txt die ersten drei Zeilen ein und gibt diese auf der Konsole aus.

```
Ladies and gentlemen, this is Mambo Number Five
One, two, three, four, five
Everybody in the car, so come on, let's ride

Process finished with exit code 0
```

Test02():

Hier wird versucht die ersten -7 Zeilen zu lesen. Natürlich lässt dies das Programm nicht zu!

```
You can't return a negative amount of lines!

Process finished with exit code 0
```

Test03():

Der User möchte die ersten 99 Zeilen aus der Textdatei haben, die jedoch aus weniger Zeilen besteht. Das Programm gibt daher einfach das ganze File aus.

```
Ladies and gentlemen, this is Mambo Number Five
One, two, three, four, five
Everybody in the car, so come on, let's ride
To the liquor store around the corner
The boys say they want some gin and juice
But I really don't wanna
Beer bust like I had last week
```

(Es geht noch weiter, aber mehr habe ich auf einmal nicht kopieren können 😂)

Test04():

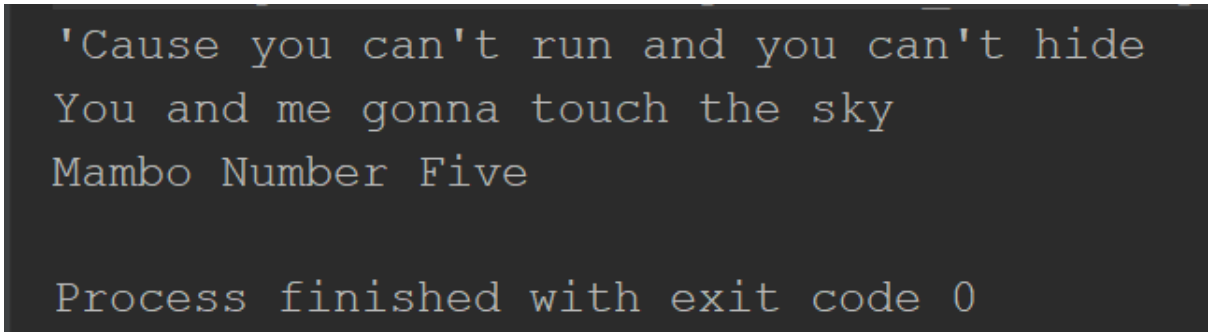
Es wird keine Zeilenzahl angegeben. Daher wird einfach das ganze File ausgegeben.

```
A little bit of you makes me your man
I do all to fall in love with a girl like you
'Cause you can't run and you can't hide
You and me gonna touch the sky
Mambo Number Five

Process finished with exit code 0
```

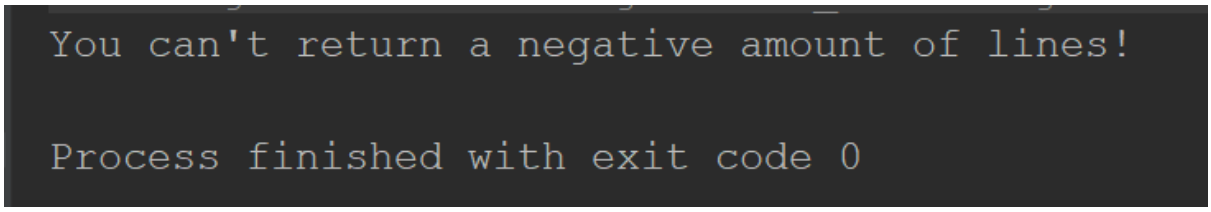
Test05():

Wie Test01(), jedoch werden hier die letzten drei Zeilen ausgegeben.



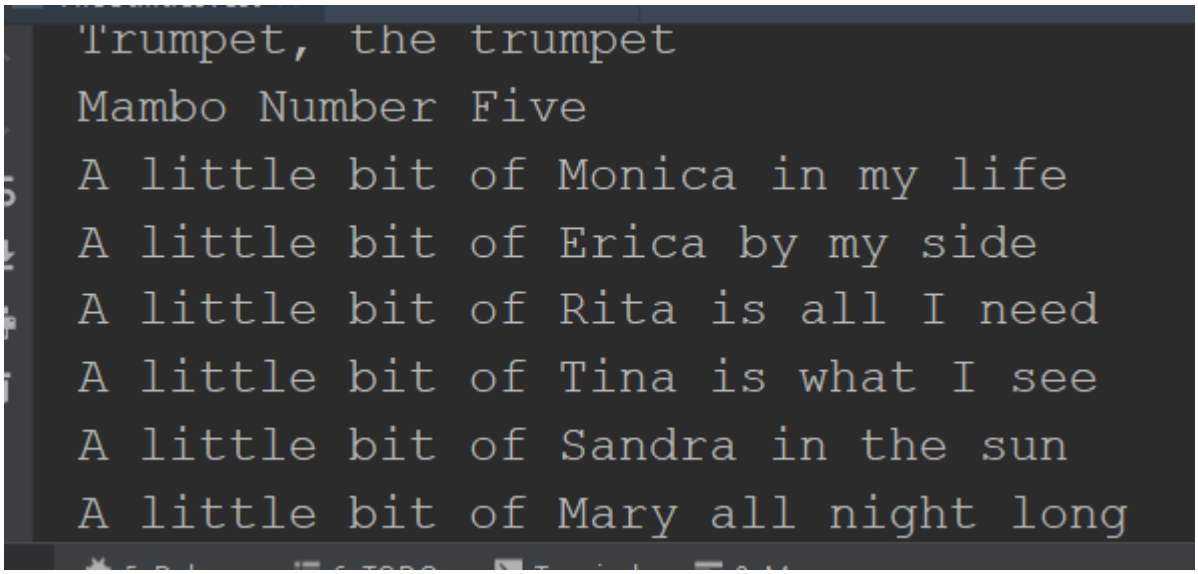
Test06():

Hier wird versucht die letzten -77 Zeilen auszugeben. Das Programm lässt dies logischerweise nicht zu!



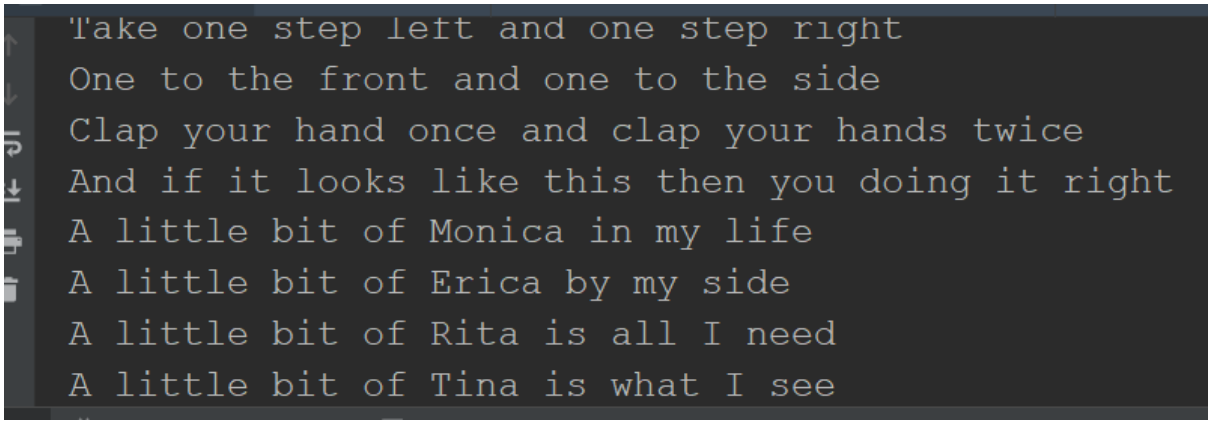
Test07():

Hier sollen die letzten 1337 Zeilen ausgegeben werden, jedoch hat das File nicht so viele. Daher wird einfach das gesamte File ausgegeben.



Test08():

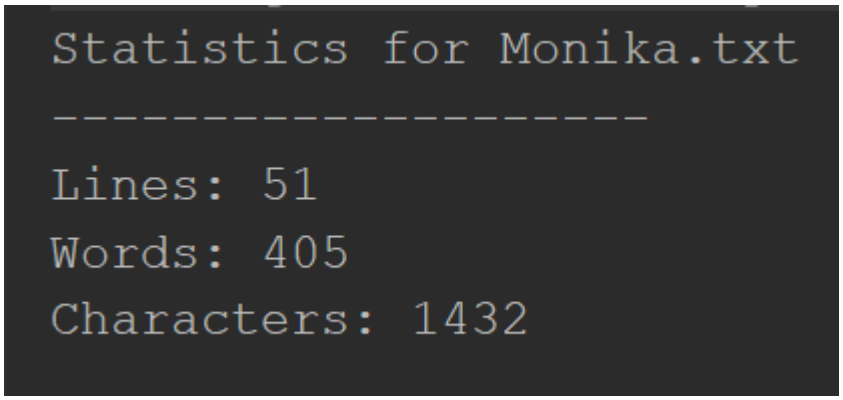
Hier wird keine Zeilenanzahl angegeben. Daher gibt das Programm einfach alles aus.



Für LOC:

Test09():

Es wird Monika.txt eingelesen und die Statistiken ausgegeben.



Test10():

Es wird ein Filename übergeben, den es nicht gibt. Dadurch wirft das Programm einen Fehler!

Test11():

Hier wurde ein leeres File übergeben.

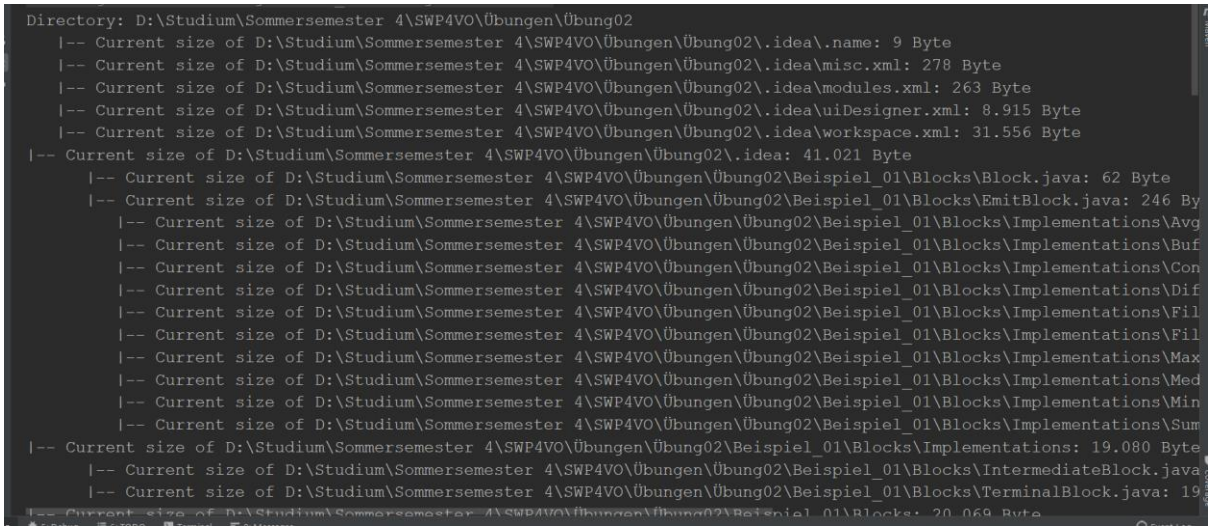
```
Statistics for IAmEmptyInside.txt
--------------------
Lines: 0
Words: 0
Characters: 0
```

Für TreeSize():

Anmerkung: Mein angegebener Pfad wird zu 99.99% nicht bei euch funktionieren. Sollte das Bedürfnis bestehen, den Testfall selbst einmal auszuführen, bitte ich, einfach einen Pfad aus Ihrem Dateisystem anzugeben. 😊

Test12():

Ein Pfad zu dem Projekt (Übung 02) wird angegeben.



Auszug aus den ersten Zeilen.

Test13():

Hier wird versucht einen Pfad zu übergeben, der ungültig ist, den es also nicht gibt. Dadurch wird eine Exception geworfen!

```
Directory: D:\IDoNotExist
The path is invalid!
Size of directory D:\IDoNotExist: 0 Byte
```

Test14():

Hier wurde ein relativ großes Verzeichnis angegeben.

```
|-- Current size of D:\Studium\Wintersemester 3\WEB Vorlesung: 820.287.284 Byte
|-- Current size of D:\Studium\Wintersemester 3: 15.781.266.837 Byte
Size of directory D:\Studium: 15.781.266.837 Byte
```

Test15():

Das Ergebnis eines leeren Verzeichnisses:

```
Directory: D:\Studium\Sommersemester 4\SWP4VO\Übungen\Übung02\IAmEmpty
Size of directory D:\Studium\Sommersemester 4\SWP4VO\Übungen\Übung02\IAmEmpty: 0 Byte
```

Der Code wird wieder ausgelagert!