
ABSCHLUSSPRÜFUNG SOMMER 2022
ENTWICKLUNG EINES SOFTWARESYSTEMS

VORGELEGT VON: SVEN BERGMANN
VORGELEGT AM: 11. MAI 2022
PRÜFLINGSNUMMER: 101 20541
AUSBILDUNGSBETRIEB: CAE GMBH

Inhaltsverzeichnis

1	Aufgabenbeschreibung	4
1.1	Aufgabenanalyse	4
1.2	Beschreibung der mathematischen Methoden	4
1.3	Konzepterstellung zur Nebenläufigkeit von Einlesen, Verarbeiten und Ausgeben	5
1.4	Einlesen und Initialisieren der Daten	6
1.5	Ausgabe gemäß Aufgabenstellung	6
2	Objektorientierter Entwurf	7
2.1	Framework	7
2.2	Problem	7
2.3	ProblemSimple	8
2.4	Utils	9
3	Änderungen zur schriftlichen Ausarbeitung	9
3.1	Umbenennungen	9
3.2	Modifikation der Klassenstruktur	9
3.3	Modifikation der Logik	9
4	Allgemeines zu Multithreading in Java	10
5	Auswertung und Interpretation	11
6	Benutzeranleitung	11
6.1	Ordnerstruktur	11
6.2	Benötigte Programme	12
6.3	Ausführen als Kommandozeilenprogramm	12
6.3.1	Ausführen der JAR	13
6.3.2	Ausführen des Python-Skripts	13
7	Zusammenfassung und Ausblick	13
A	UML-Diagramme	14
A.1	Klassendiagramme	14
A.1.1	Main	14
A.1.2	Pakete	16
A.1.3	Klassen im Paket „framework“	20
A.1.4	Klassen im Paket „problem“	25
A.1.5	Klassen im Paket „problemsimple“	31
A.1.6	Klassen im Paket „utils“	33
A.2	Nassi-Shneidermann Diagramme	35
A.3	Sequenzdiagramme	37

1 Aufgabenbeschreibung

1.1 Aufgabenanalyse

Eine Firma möchte einen optischen Autokorrelator¹ bauen, welcher mit einem Laser, halbdurchlässigen und voll reflektierenden Spiegeln, einem Kristall, einem Filter und einem Detektor funktioniert. Der Detektor misst hierbei die Autokorrelationsfunktion des Eingangssignals, die Aufschluss über die Breite des Laserpulses oder die Ursprungsfrequenz gibt. Die Messdaten werden allerdings vom Detektor permanent und nebenläufig geliefert, was mit einem Thread modelliert werden soll, welcher mit 20 Hz, also alle 0,05 Sekunden eine neue Messdatei zur Verfügung stellt und die alte überschreibt. Der Lese-Thread ist hierbei Thread A und läuft ständig weiter, indem die Daten "0.txt" bis "9.txt" immer wieder eingelesen werden und den aktuellen Datensatz überschreiben. Der Thread B soll dann diese Messdaten weiterverarbeiten und mit den später erklärten mathematischen Methoden die Funktion modellieren, bzw. die Messwerte umrechnen und glätten. Schließlich wird die fertige Berechnung dann an Thread C weitergegeben, um die Daten in seinem auch später erklärten Format in eine Datei auszugeben, welche als Präfix "out" besitzt und ansonsten gleich heißt. Das Programm wird dann beendet, wenn alle Messdaten verarbeitet wurden. Es kann durchaus vorkommen, dass eine Messdatei häufiger eingelesen wird. Diese muss dann allerdings nicht erneut verarbeitet werden, es wird dann einfach auf eine noch nicht verarbeitete Datei gewartet.

1.2 Beschreibung der mathematischen Methoden

Als erstes werden alle \hat{x} -Werte in Pikosekunden umgerechnet, was mit der Formel

$$\hat{x}_k = \frac{\tilde{x}_k}{2^{18} - 1} \cdot 266,3 - 132,3$$

passieren soll. Zur Zeitersparnis und Verbesserung der Genauigkeit wurde hierbei eine Umstellung sinnvoll sein, sodass der Bruch $\frac{266,3}{2^{18}-1}$ nicht immer wieder erneut berechnet werden muss. Also wird es eine Variable α geben, welche diesen Bruch beschreibt und die Formel wird zu

$$\hat{x}_k = \alpha \cdot \tilde{x}_k - 132,3$$

Des Weiteren werden noch alle y-Werte normiert, sprich jeder Wert wird durch das Maximum aller y-Werte geteilt.

Danach sollen die Daten geglättet werden, indem der sog. gleitende Mittelwert berechnet werden soll. Die Formel hierfür ist

$$x_k = \frac{1}{n} \sum_{i=0}^n x_{k-\tau+i}$$

mit $\tau = \frac{n-1}{2}$ und

$$n = \begin{cases} \lfloor 0,002 \cdot N \rfloor - 1 & \text{für } \lfloor 0,002 \cdot N \rfloor \text{ gerade} \\ \lfloor 0,002 \cdot N \rfloor & \text{für } \lfloor 0,002 \cdot N \rfloor \text{ ungerade} \end{cases}$$

¹Link: <https://de.wikipedia.org/wiki/Autokorrelator>

wobei n die Größe des Mittelungsfensters beschreibt, ungerade ist und 0,2% von N entspricht. Es soll zudem noch geeignete Werte für $k < \tau$ und $k > N - 1 - \tau$ bestimmt werden, sprich den linken und rechten Rand der Messungen. Das Symbol $\lfloor x \rfloor$ heißt untere Gaußklammer und beschreibt, wie auf ganze Zahlen gerundet werden soll.

Als nächstes wird die obere Einhüllende bestimmt, welche die Autokorrelationsfunktion oben komplett einschnürt. Da dies aber numerisch sehr aufwändig ist, reicht es aus, diese Einhüllende einfacher zu approximieren und zwar indem zuerst von links beginnend jedem Positionswert der zuletzt höchste Intensitätswert zugeordnet wird, solange bis man am Maximum ist, dann wird das gleiche Verfahren von rechts wiederholt.

Zuletzt wird noch die Pulsbreite b berechnet, welche sich über den Abstand der beiden Punkte L und R berechnet. Diese sind Punkte auf der oberen Einhüllende, wobei die Gerade die durch L und R geht, genau auf der Hälfte der Grundlinie und des Maximums liegt. Die Grundlinie stellt dabei die mittlere Höhe des äußersten linken Prozents der Intensitätswerte dar. Es müssen also zuerst die gemittelten Werte der ersten Berechnung aufsteigend sortiert werden, wobei dann geschaut wird, welche Werte alle im Intervall $[0; 0,01]$ liegen. Diese werden dann ebenfalls gemittelt und der daraus resultierende Wert beschreibt die Grundlinie.

1.3 Konzepterstellung zur Nebenläufigkeit von Einlesen, Verarbeiten und Ausgeben

Bei dem gegebenen Problem könnte ganz strikt nach dem IPO (Input-Process-Output) oder EVA (Eingabe-Verarbeitung-Ausgabe) Prinzip gearbeitet werden, wobei jedoch nur genau ein Thread für das Liefern der Daten, also den Input zuständig ist. Hierfür könnte ein Interface bereitgestellt werden, welches einen Thread oder Prozess implementiert. Dieses Interface stellt dann eine default Methode bereit, die in einem Intervall von 20 Hz immer wieder aus der zu implementierenden read Methode der Daten holt und weitergibt. Der Thread B wird dann, wie auch der Thread C, so lange laufen, wie noch Daten nicht verarbeitet wurden. Wenn Thread A also eine Datei bereitstellt, startet Thread B und berechnet das Ergebnis, welches dann vom Thread C geschrieben werden kann. Thread A, B und C können somit gleichzeitig ausgeführt werden, da die Verarbeitung unabhängig voneinander passiert. Für Thread B würde also ebenfalls ein Interface bereitgestellt werden, welches einen Thread implementiert und die Methode process mit den Parametern aus Thread A bereitstellen muss. Der Thread C sieht dann ebenfalls ein Interface vor, welches einen Thread implementiert und die Methode write bereitstellt. Falls Thread C alle Messdaten geschrieben hat, wird Thread A gestoppt und damit auch Thread B. Nachdem alle Threads gestoppt wurden, wird auch das Programm beendet.

Die Klasse AKF beinhaltet einige Arrays, welche immer die Größe N haben werden und von denen eventuell nicht alle immer gebraucht werden. Laut Aufgabenstellung stellt der Lesethread (Thread A) alle 0,05 Sekunden einen neuen Datensatz bereit, was wahrscheinlich sinnvoller ist als "push" Methode zu implementieren. Des Weiteren habe ich das so verstanden, dass von allen Threads nur jeweils eine Instanz gleichzeitig laufen kann, was die Überschreibung und Wiederholung der Datensätze erklären würde, jedoch in einem Produktivsystem nicht allzu sinnvoll wäre, da ansonsten Datensätze verloren

gehen. Der Thread B läuft also nur einmal und verarbeitet auch nur einen Datensatz gleichzeitig, bevor der nächste geholt oder gepusht wird. Eventuell wäre daher auch das Observer-Observable Pattern sinnvoll, da der Thread B informiert werden muss, falls Thread A Daten bereitstellt. Genau das Gleiche wäre dann natürlich auch für Thread B und C sinnvoll.

Das Pattern könnte dann so abgewandelt werden, dass Thread A ein Observable implementiert, worin der Observer Thread B registriert ist und über die notify Methode den neuen Datensatz bekommt. Falls der Thread B allerdings noch arbeitet, wird das einfach ignoriert. Thread C wird denn selbst auch als Observer im Observable Thread B registriert und bekommt so die fertig verarbeiteten Daten. Als Abbruchbedingung könnte eventuell noch der Thread C als Observable implementiert werden und Thread A als Observer registrieren, um alle bereits geschriebenen Dateinamen mitzuteilen. Weiterhin habe ich in dem UML Diagramm alles generisch mit dem Type T gelassen, da am besten ein Paar aus Dateinamen und AKF übergeben werden sollte, allerdings braucht Thread B zum Beispiel nicht die komplette Klassenstruktur, weshalb der generische Typ hier noch sinnvoll ist. Theoretisch braucht Thread C dann eben auch nur zwei Strings als Übergabe.

1.4 Einlesen und Initialisieren der Daten

Das Einlesen der Daten passiert über die read Methode in Thread A. Besonders große Schwierigkeiten gibt es hierbei nicht, da alle Zeilen mit einem „#“ startend ignoriert werden können. Die weiteren Zeilen bezeichnen die Messwerte, welche mit „\t“ getrennt sind. Hierbei ist die erste Zahl der y-Wert, also die Intensität des Signals und die zweite Zahl der x-Wert, also die Position des Spiegels. Falls während des Einlesevorgangs eine Exception geschmissen wird, wird der Vorgang einfach abgebrochen und null zurückgegeben. Das habe ich so realisiert, da der ThreadA die Daten schnell lesen muss und keine lange Fehlerbehandlung machen kann, falls diese korrupt sind. Die Werte von x und y werden zuerst in Paare eingelesen, welche in einer Liste gespeichert werden. Laut Aufgabenstellung sind dies immer positive, ganzzahlige Werte. Diese Liste von Paaren wird dann in zwei Arrays, xStart und yStart, umgewandelt und mit dem Dateinamen in ein Data Objekt gegeben und von der Methode zurückgegeben.

1.5 Ausgabe gemäß Aufgabenstellung

Die Ausgabe der fertigen Autokorrelationsfunktion erledigt dann ThreadC. Hierfür habe ich eine Methode in der Klasse AKF geschrieben, welche einen formatierten String für die Ausgabe zusammenbaut und zurückgibt. Da in der Klasse der Wert für die Pulsbreite, den IndexL und den indexR gesetzt ist, ist die erste Zeile kein Problem. Die weiteren Zeilen werden dann mit den Arrays „xTransformiert“, „yNormiert“ und „obereEinhuellende“ zusammengebaut.

Wenn der String fertig ist, wird das Ganze in eine Datei mit dem Namen „out“ + „fileName“ aus der Klasse AKF geschrieben.

2 Objektorientierter Entwurf

2.1 Framework

Das Paket Framework beinhaltet alle wichtigen Interfaces und eine abstrakte Klasse zur Lösung des gegebenen Problems. Somit wird die geforderte Austauschbarkeit des Threads A erreicht, man kann aber auch alle anderen Threads austauschen.

Das vorher genannte Observer-Observable-Pattern habe ich mit einem Interface „Observer“ und einer abstrakten Klasse „Observable“ vorgegeben. Der „Observer“ ist generisch und muss die Methode „void update(T t)“ bereitstellen, welche vom „Observable“ aufgerufen werden kann. Ebenfalls generisch gestaltet sich die „Observable“ Klasse, die eine Liste von „Observern“ hält. In diese Liste kann man über die Methode „public void registerObserver(Observer observer)“ einen „Observer“ hinzufügen. Diese werden dann über die Methode „public void notifyObservers(T t)“ über ein verändertes Objekt informiert.

Das Threading ist durch die drei Interfaces „Runnable“, „WriteRunnable“ und „ProcessRunnable“ implementiert, welche selbst das Interface „Runnable“ erweitern. Der Lesethread soll durch das Interface „ReadRunnable“ implementiert werden, welches die Methode „T read(Path pathToFile)“ bereitstellt und generisch ist. „ProcessRunnable“ stellt das Interface für den Verarbeitungsthread dar und gibt die Methode „R process(P p)“ vor, wobei R der Typ des Ergebnisparameters ist und P der Type des Eingabeparameters. Schlussendlich wird der Schreibthread durch das Interface „WriteRunnable“ dargestellt, wobei man die Methode „boolean write(T t)“ implementieren muss, die einen generischen Parameter schreiben soll.

2.2 Problem

Problem beinhaltet alle Klassen zur (optimierten) Lösung des gegebenen Problems, welche unter anderem von der Klasse und den Interfaces aus „Framework“ erben.

Die Klasse „Algorithms“ beinhaltet alle vier Algorithmen, welche über die Methode „public static AutoKorrelationsFunktion solve(Data data)“ ausgeführt werden können. Die einzelnen Algorithmen sind daher auch private, sodass Zugriffe von Außen nicht möglich sind. Als Input Parameter nimmt die Funktion ein Objekt der Klasse „Data“, welches als Record² implementiert ist und den Namen der Datei, das Array der eingelesenen x-Werte und das Array der eingelesenen y-Werte beinhaltet. Diese Werte werden dann intern durch die Algorithmen zu einem neuen Objekt vom Typen „AutoKorrelationsFunktion“ zusammengebaut. Dieser Typ ist ebenfalls als Record realisiert und hält den Dateinamen, die Pulsbreite, den indexL, den indexR, das double-Array der transformierten x-Werte, das double-Array der normierten y-Werte und das double-Array der Werte der oberen Einhüllenden.

Die Klasse „ThreadA“ implementiert das Interface „Runnable<Data>“, erweitert die abstrakte Klasse „Observable< Pair<Data, Integer> >“ und stellt eben genau den Thread A aus der Aufgabenstellung dar. Als Konstruktor-Parameter wird der relative Pfad zum Ordner der Eingabedateien und der sleepTime Wert als long angegeben. Die Methode

²<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html>

„public void run()“ des Runnable Interfaces wird so überschrieben, dass zuerst alle Pfade im Eingabeordner ausgelesen werden und dann eine Endlosschleife beginnt. In dieser Endlosschleife wird immer wieder durch die Pfade iteriert, die jeweilige Datei eingelesen und alle Observer mit einem Paar aus dem erstellten Data Objekt und der Anzahl der Daten im Ordner weitergegeben. Danach wartet der Thread so lange, wie es in sleeptime angegeben ist, laut Aufgabenstellung 0,05s, wonach der Schleifendurchlauf weitergeht, oder von vorne beginnt. Die Methode zum Einlesen der Daten in das Data Objekt ist in Einlesen und Initialisieren der Daten beschrieben.

Klasse „ThreadB“ erweitert die abstrakte Klasse „Observable<AutoKorrelationsFunktion>“, implementiert die Interfaces „Observer<Pair<Data, Integer>>“ und „ProcessRunnable<Data, AutoKorrelationsFunktion>“ und stellt den Thread B der Aufgabenstellung dar. Der Konstruktor fordert „maxPoolSize“ und „maxQueueSize“ als Parameter und erstellt dadurch einen neuen ThreadPoolExecutor³ mit den gegebenen Parametern. Dieser Executor wird dann für das Master-Worker Pattern genutzt, indem die von „ThreadA“ eingelesenen Daten an verfügbare Worker zur Weiterbearbeitung verteilt werden. Falls kein Worker mehr zur Verfügung steht, wird die aktuelle Datei verworfen und die nächste eingelesene Datei bearbeitet, falls bis dahin wieder Worker frei sind. Sobald eine Datei fertig bearbeitet ist und das Objekt vom Typen „AutoKorrelationsfunktion“ erstellt wurde, werden wieder alle Observer mit dem neuen Objekt informiert.

„ThreadC“ implementiert die Interfaces „Observer<AutoKorrelationsFunktion>“ und „WriteRunnable<AutoKorrelationsFunktion>“ und stellt den Thread C der Aufgabenstellung dar. Der Konstruktor nimmt hier nur den Pfad des Ausgabeordners. In der Methode „public void update(AutoKorrelationsFunktion akf)“ wird für jedes neu geschickte Objekt ein Runnable erstellt, welches mit „CompletableFuture.supplyAsync(()->this.write(akf))“ ausgeführt und in ein HashSet gepackt wird. Somit kann jede Schreiboperation asynchron ausgeführt werden. Falls der Thread B dann ein null Objekt im Update übergibt, so wird auf jedem Worker „join“ aufgerufen, also auf die Ausführung gewartet und danach das Programm beendet. Die Methode „public boolean write(AutoKorrelationsFunktion akf)“ testet, ob der Ordner existiert, erstellt einen neuen falls nicht und testet, ob die Ausgabedatei existiert. Falls ja wird das in den Logger geschrieben und die Datei wird mit dem Inhalt der Funktion „public String getOutputString()“ in AutoKorrelationsFunktion überschrieben, welche in Ausgabe gemäß Aufgabenstellung beschrieben wurde.

2.3 ProblemSimple

In „problemsimple“ findet sich eine alternative Implementierung des gegebenen Problems wieder, es wurden allerdings nur die Klassen „ThreadB“ und „ThreadC“ neu geschrieben, alle anderen Klassen werden weiter benutzt. Wie der Name schon sagt, wird hier naiv und mit einer einfachen Fehlerbehandlung an das Problem herangegangen. Während dieser Ausführung existieren nur genau diese drei Threads A,B und C, wobei das Master-Worker Pattern nicht implementiert ist.

³<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html>

„ThreadB“ erweitert nun die abstrakte Klasse „Observable<Pair<Autokorrelationsfunktion, Integer>>“ und implementiert die Interfaces „Observer<Pair<Data, Integer>>“ und „ProcessRunnable<Pair<Data, Integer>, AutoKorrelationsFunktion>“. Dies ist nötig, da nun die Anzahl der zu bearbeitenden Elemente an ThreadC weitergeschickt werden kann.

„ThreadC“ implementiert nun die zwei Interfaces „Observer<Pair<AutoKorrelationsFunktion, Integer>>“ und „WriteRunnable<AutoKorrelationsFunktion>“, um die Anzahl der Daten geschickt bekommen zu können. Die Methode „public boolean write(AutoKorrelationsFunktion akf)“ ist gleichgeblieben.

2.4 Utils

Das Paket „utils“ stellt zwei Klassen, „CmdLineParser“ und „Pair“. Der „CmdLineParser“ bekommt als Konstruktorparameter das String-Array der Kommandozeilenargumente des Programmaufrufs übergeben und versucht diese auszulesen und zu setzen. Mit den Getter Methoden können diese dann z. B. von der Main-Klasse abgerufen werden und in die Threads übergeben werden. Des Weiteren existiert noch das Record „Pair“, welches ein nicht veränderliches generisches Objekt von zwei Typen K und V darstellt.

3 Änderungen zur schriftlichen Ausarbeitung

3.1 Umbenennungen

Es sind fast alle der Benennungen so wie in der schriftlichen Ausarbeitung geblieben, es wurden nur alle Benennungen an das Deutsche angepasst. Die Java-spezifischen Benennungen wie „z. B. getAttribut oder setAttribut“ sind geblieben. Zudem wurden natürlich noch die Umlaute umschrieben.

3.2 Modifikation der Klassenstruktur

In der schriftlichen Ausarbeitung wurden noch keine Paketnamen angegeben. Wie bereits genannt, existieren die Pakete „framework“, „problem“ und „utils“. Des Weiteren wollte ich jede Thread Klasse sowohl von Observable, Observer, als auch einem der drei Threadinterfaces erben lassen. Hier kommt aber das Diamond-Problem zum tragen und Java verbietet daher das Erben mehrerer Klassen. Ich habe mich also für mehrere Interfaces entschieden und benutze jetzt das „Runnable“ Interface anstatt die Thread Klasse. Ansonsten ist die Klassenstruktur im Groben gleich geblieben.

3.3 Modifikation der Logik

Wie schon gesagt werden mittlerweile Runnables⁴ anstatt Threads⁵ genutzt. Dies hat den Vorteil, dass nun die Klassen Thread A, B und C von mehreren Interfaces wie Runnable

⁴<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html>

⁵<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.State.html>

erben können und zusätzlich die abstrakte Klasse „Observable“ extenden können. Jede Threadklasse überschreibt also nur die „run()“ Methode und benutzt die Methoden des Observer-Observable-Patterns. Eine Weitere Modifikation besteht darin, dass nun der Thread A den Thread B als Observer registriert hat und der Thread B den Thread C als Observer. Das heißt, dass die Threads nun in einer Reihe verkettet sind und nichtmehr wie bei der schriftlichen Ausarbeitung in einem Ring.

Es wurden des Weiteren zwei mögliche Implementierungen dargestellt, einmal mit dem Master-Worker Pattern und einmal ohne. Die Implementierung des Master-Worker Patterns befindet sich im Paket „problem“, während die einfache Implementierung im Paket „problemsimple“ zu finden ist. Beide Implementierungen nutzen die gleiche Klasse ThreadA, unterscheiden sich aber in den Klassen ThreadB und ThreadC und vorallem auch in der Abbruchbedingung des Programms.

In „problem“ wird das ganze Programm erst von Thread C beendet, falls dieser in der „update(T t)“ Methode null zurück bekommt, da nur der ThreadB weiß, wann keine Daten mehr eingelesen, bzw. verarbeitet werden. Sowohl ThreadB, als auch ThreadC implementieren das Master-Worker Pattern und bei Thread B kann zusätzlich angegeben werden, wie groß der Threadpool sein soll und wie viele Objekte in die Warteschlange gepackt werden dürfen. ThreadC schreibt einfach alle gepushten Daten asynchron, da davon ausgegangen wird, dass die Verarbeitung länger dauert, als das Schreiben.

In „problemsimple“ wird nur mit genau diesen drei Threads gearbeitet und keinen Weiteren. Um dies zu realisieren, gibt es in beiden Klassen nun eine Variable „processing“ oder „writing“, welche beide vom Typen AtomicBoolean⁶ sind, da diese Variablen threadsafe sind. Falls nun der ThreadA zu schnell Daten pusht, sodass ThreadB in einem bearbeitenden Zustand ist, so überspringt ThreadB einfach diese Datei und bearbeitet die nächste, sobald er fertig ist. ThreadC macht das ähnlich, woraus auch resultiert, dass ThreadC die Abbruchbedingung in diesem Fall verwalten muss. Die Anzahl der Input Dateien wird also von ThreadA über ThreadB nach ThreadC geschleuft und in ThreadC wird dann immer geprüft, ob die aktuelle Datei schon geschrieben wurde und ob schon alle Daten geschrieben wurden. Ist dies der Fall, wird das Programm beendet.

4 Allgemeines zu Multithreading in Java

In Java kann Multithreading auf verschiedene Arten realisiert werden. Das etwas ältere, aber daraus noch genutzte Konzept besteht darin, die Thread Klasse zu erweitern und die Methoden, vorallem die „run()“ Methode, zu überschreiben. Der Vorteil hierbei ist, dass von Haus aus Basisfunktionalitäten, wie „yield()“, oder „sleep()“ bereitgestellt werden. Eine weitere Methode ist es, das Interface „Runnable“ zu implementieren, was den Vorteil mit sich bringt, dass man mehrere Interfaces implementieren kann, aber nur eine Klasse erweitern kann. „Runnable“ wird auch von der Klasse „Thread“ implementiert, stellt aber nur Basisfunktionalitäten bereit, wie die „run()“ Methode. Ein Objekt vom Typen „Runnable“ kann aber dafür genutzt werden dieses von mehreren Threads ausfüh-

⁶<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/AtomicBoolean.html>

ren zu lassen. Das Ausführen der beiden Objekte verhält sich auch unterschiedlich. Ein Objekt des Types „Thread“ kann einfach mit „new Thread“ erstellt werden und dann mit „Thread.start()“ gestartet werden, wodurch jedes Mal eine neue Instanz eines Thread Objektes erstellt wird. „Runnables“ hingegen werden entweder einem vorher erstellten „new Thread()“ Objekt als Parameter mitgegeben und dann gestartet, oder aber einem Executor Service übergeben, welcher entweder selbst erstellt und verwaltet werden kann, oder einem bereits vorgegebenen Service zugeordnet werden kann. Vorgegebene Services sind zum Beispiel der „newCachedThreadPool()“, welcher neue Threads erstellt, solange diese gebraucht werden und alte wieder nutzt, oder der „newFixedThreadPool(int nThreads)“, welcher eine fixe Anzahl an Threads nutzt, um die Aufgaben auszuführen und ansonsten eine Queue benutzt, in welcher die Aufgaben gespeichert und dann nach und nach abgearbeitet werden. Je nach Programm oder Problemstellung kann man also diese Services nutzen. In meinem Programm nutze ich in der „Main“ zur Ausführung der selbstgeschriebenen Thread Klassen, welche das Interface Runnable implementieren, einen newFixedThreadPool mit 3 Threads. Diese Runnables werden dann von diesem Threadpool verwaltet. Falls man nun nicht nur wie in dieser Aufgabenstellung laufende Threads haben möchte, sondern auch ein Ergebnis aus diesen Threads erwartet, mit dem dann weitergemacht werden soll, kann man die sogenannten „CompletableFutures“ nutzen, welche über den Funktionsaufruf „CompletableFuture.supplyAsync(<Runnable>)“ erstellt und asynchron ausgeführt werden. Mit dem „CompletableFuture“ Objekt kann dann mit Methoden wie z. B. „thenApply()“ weitergearbeitet werden. Diese werden in dem ThreadC der Methode mit der Implementierung des Master-Worker Patterns genutzt. Um das Ganze allerdings etwas lebensechter zu machen habe ich in ThreadB einen eigenen ThreadPoolExecutor erstellt, welcher die gewünschte Anzahl an Threads und die gewünschte Größe der Warteschlange setzt. Somit kann man gut simulieren, was passiert, wenn keine Worker mehr verfügbar sind und die Queue zudem auch voll ist.

5 Auswertung und Interpretation

6 Benutzeranleitung

Generell befindet sich die Gesamtdokumentation der Klassen und Methoden als javadoc im „docs“ Ordner.

6.1 Ordnerstruktur

An sich sollten die fertig gebaute „.jar“ Datei zusammen mit dem ausführbaren Python Skript in einem Ordner liegen. Die Testbeispiele sollten ebenfalls auf der gleichen Ebene in einem Ordner vorhanden sein, wobei der Ordnername beim Programmstart oder in dem Skript angegeben werden kann. Der Ordner für alle Ausgabe Dateien wird dann ebenso in dieser Ebene erstellt. Die Ausgabedateien werden pro Eingabedatei jeweils in eine Datei namens „out<Name der Eingabedatei>.txt“ im output Ordner gespeichert, welcher entweder über die Programmzeile vorher angegeben wird, oder einfach „output“ heißt.

6.2 Benötigte Programme

Für die Ausführung der „.jar“ Datei benötigt der Zielrechner zwingend eine Installation des Open JDK 17⁷, da das Java-Sprachlevel auf 17 gesetzt wurde. Um das Python Skript auszuführen wird eine Python Installation gebraucht, wobei ich die Version 3.10⁸ benutzt habe. Um das Projekt zu bauen, wird Gradle⁹ genutzt.

6.3 Ausführen als Kommandozeilenprogramm

Es gibt zwei Wege das Programm auszuführen, einmal direkt über Java und einmal über das Python Skript, welches intern die „.jar“ Datei aufruft.

Das Programm nimmt Argumente entgegen, welche sind:

- -inputfolder {„Name des Ordners mit den Eingabedateien“}, default: „input“
- -outputfolder {„Name des Ordners, wohin die Ausgabedateien geschrieben werden“}, default: „output“
- -log {„true“ oder „false“ oder „file“}, default: „file“
- -loglvl {„warning“ oder „info“}, default: „all“
- -poolsize {„Anzahl der gewünschten Threadpools“}, default: „1“
- -sleep {„Zeit in Millisekunden mit welcher der ThreadA Daten pusht“}, default: „50“
- -queuesize {„Größe der Schlange der Tasks“}, default: „1“
- -mw {„true“ oder „false“}, default: „false“

Man kann jede dieser Optionen setzen, muss das aber nicht tun. Falls eine Option nicht gesetzt wird, wird der default Wert angenommen.

Die Namen der Ordner werden immer relativ zum derzeitigen Ordnerpfad ausgewertet. Die Logoption beschreibt das Verhalten des Loggers, beziehungsweise wohin die Logs geschrieben werden. Der LogLevel setzt das Level der geloggten Nachrichten, sprich welche Nachrichten tatsächlich geloggt werden. Poolsize gibt an, wie groß die Anzahl der gewünschten Threads sein soll, die der ThreadPool von Thread B maximal bereitstellt. Sleep gibt die Zeit in Millisekunden an, die der ThreadA wartet, nachdem dieser Daten gepusht hat und QueueSize beschreibt die Größe der Schlange des Services des ThreadsB. Die Option „-mw“ gibt an, ob das Programm die Thread Klassen mit dem intern implementierten Master-Worker Pattern nutzen soll, oder die Threads ohne das Master-Worker Pattern.

⁷<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

⁸<https://www.python.org/downloads/release/python-3100/>

⁹<https://gradle.org/>

6.3.1 Ausführen der JAR

Um die „jar“ ausführen zu können, muss zumindest unter Windows der Pfad zum JDK 17 in den Umgebungsvariablen gesetzt sein. Mit einem Doppelklick auf die Datei wird der Code mit Standardargumenten ausgeführt. Ansonsten könnte der Benutzer auch über die Kommandozeile gehen und die Programmargumente selbst setzen. Das würde dann beispielsweise so aussehen:

```
java -jar IHK_Abschlusspruefung.jar -inputfolder input -outputfolder  
output -log file -loglvl warning -poolsize 1 -sleep 50 -queuesize 1
```

6.3.2 Ausführen des Python-Skripts

Der Benutzer kann zusätzlich auch noch das Python-Skript „execute_gro_pro.py“ ausführen, um das Programm mit den Standardargumenten zu starten. Hierfür ist eine Installation von Python notwendig, sowie die Verlinkung zur Umgebungsvariablen. Der Kommandozeilencode sieht dann beispielsweise so aus:

```
python execute_gro_pro.py
```

7 Zusammenfassung und Ausblick

A UML-Diagramme

A.1 Klassendiagramme

A.1.1 Main

Abbildung 1: Main

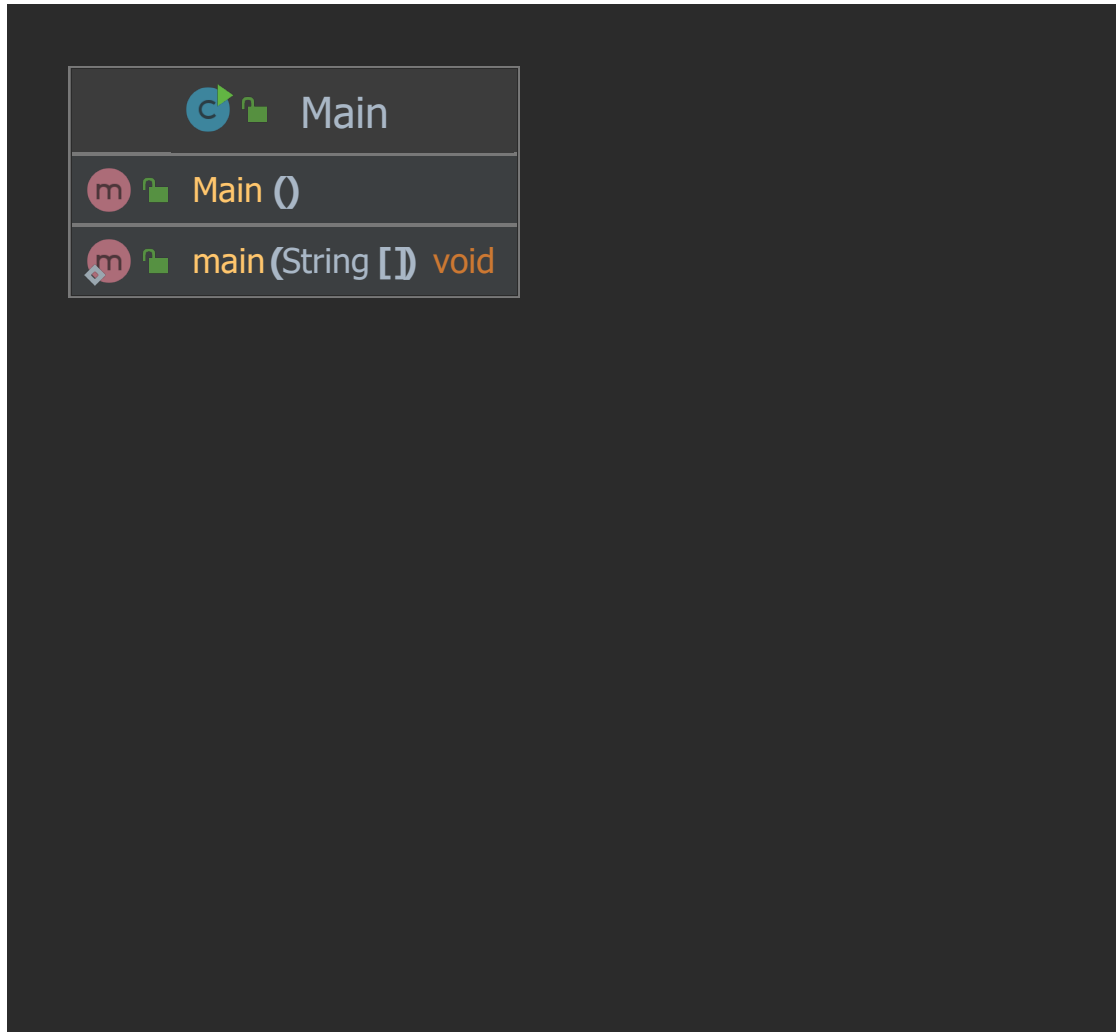
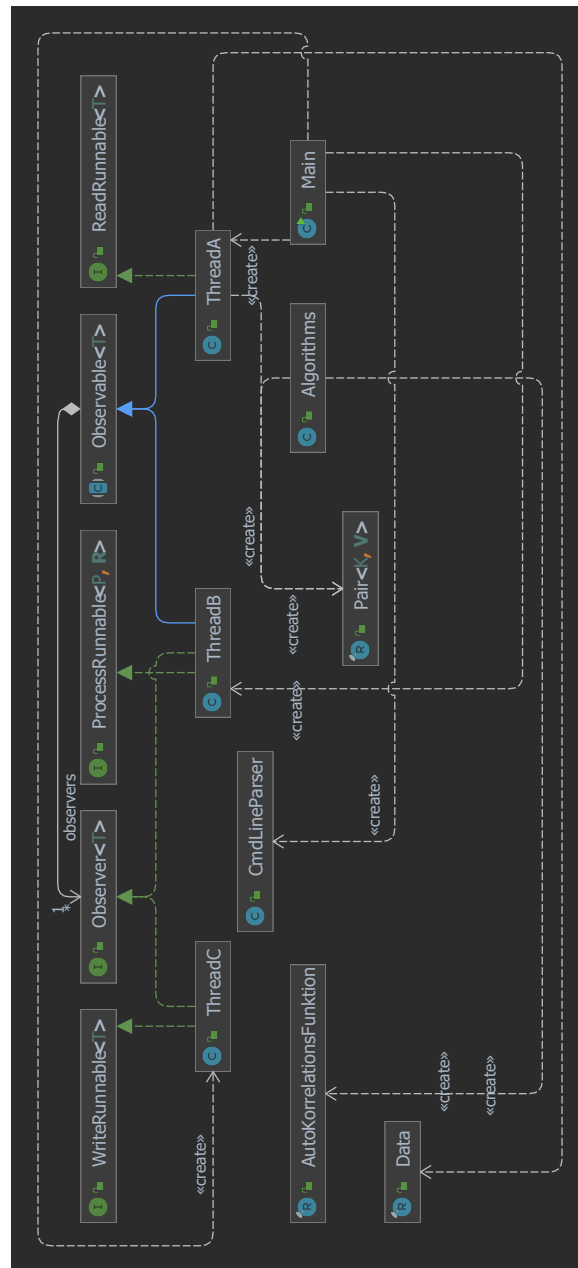


Abbildung 2: com.de.cae



A.1.2 Pakete

Abbildung 3: framework

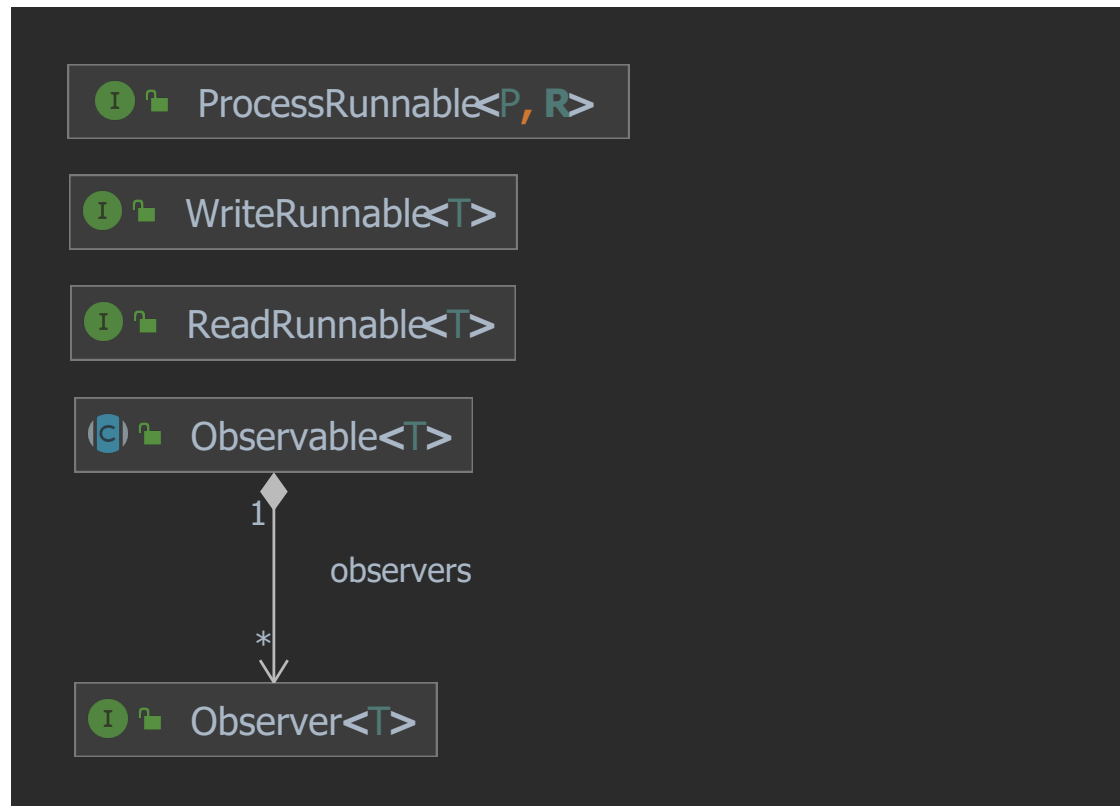


Abbildung 4: problem

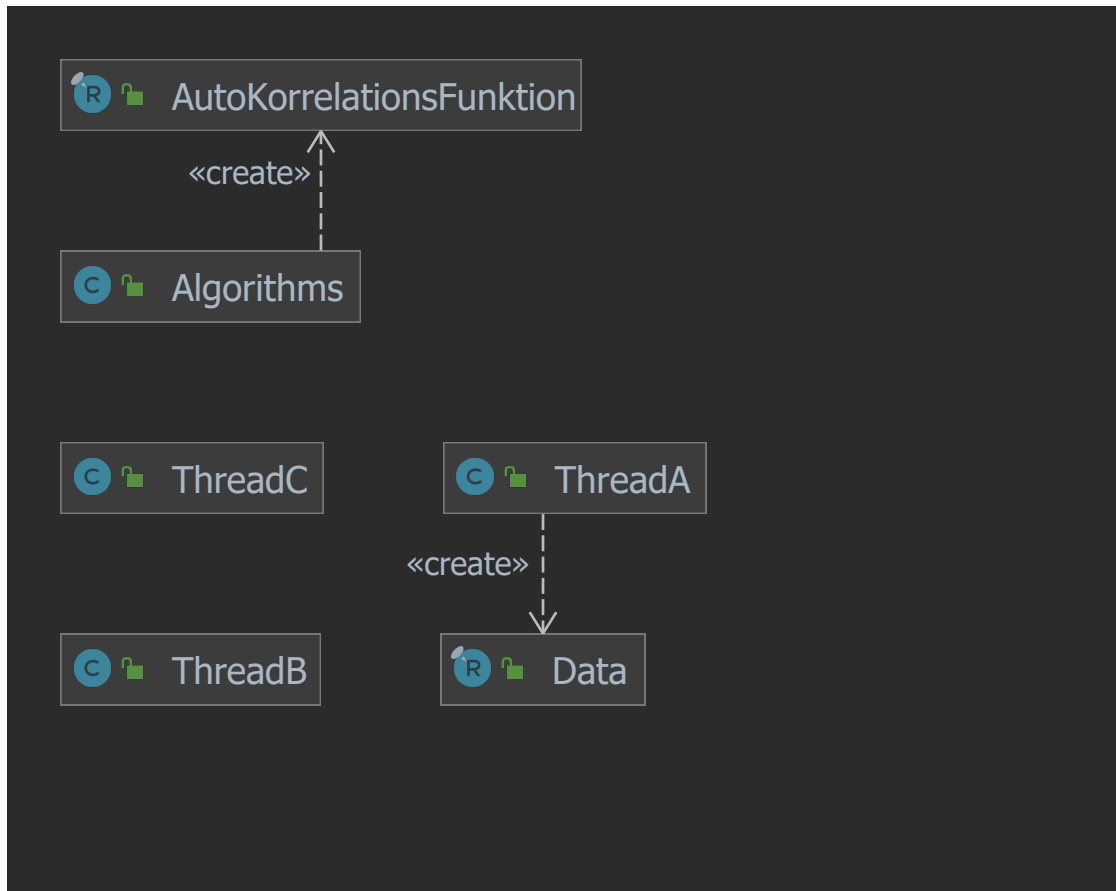


Abbildung 5: problemsimple

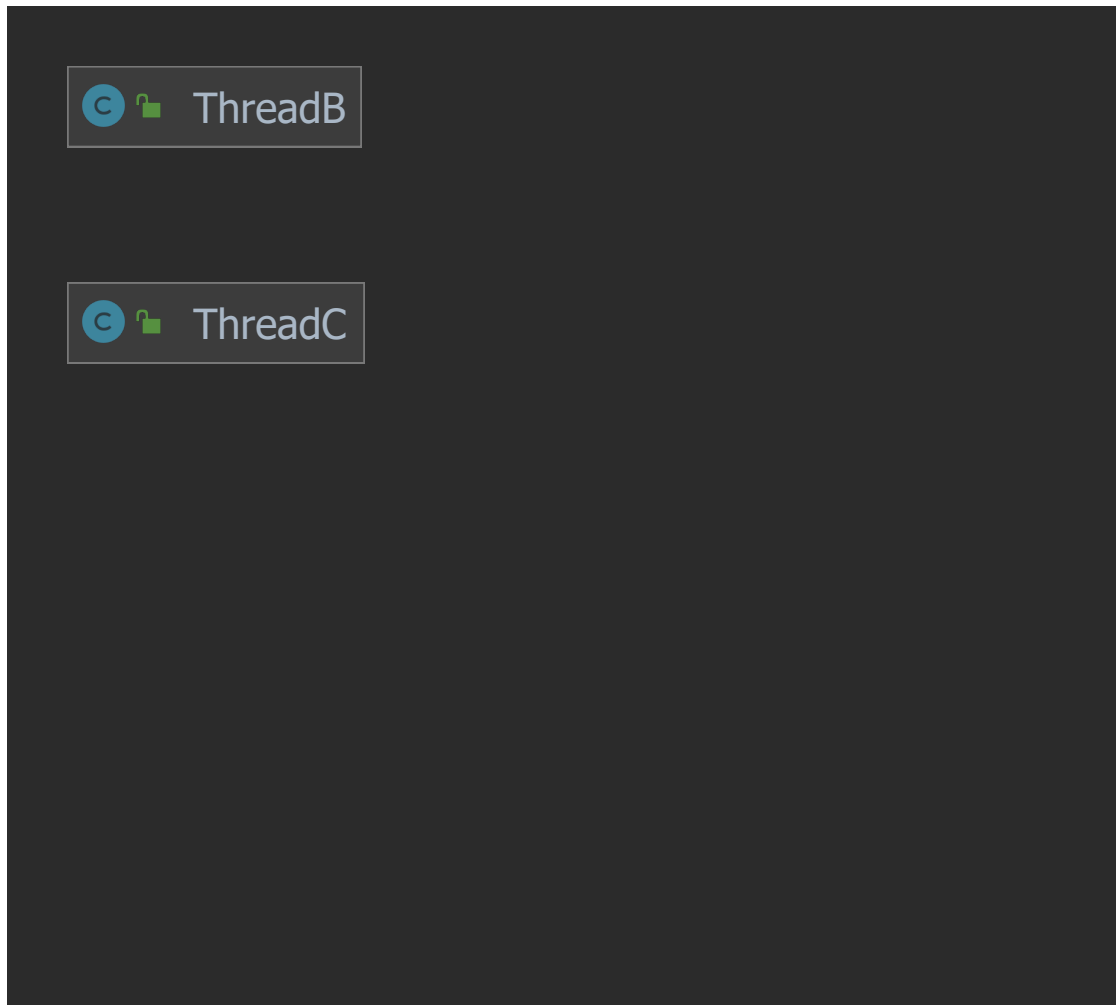
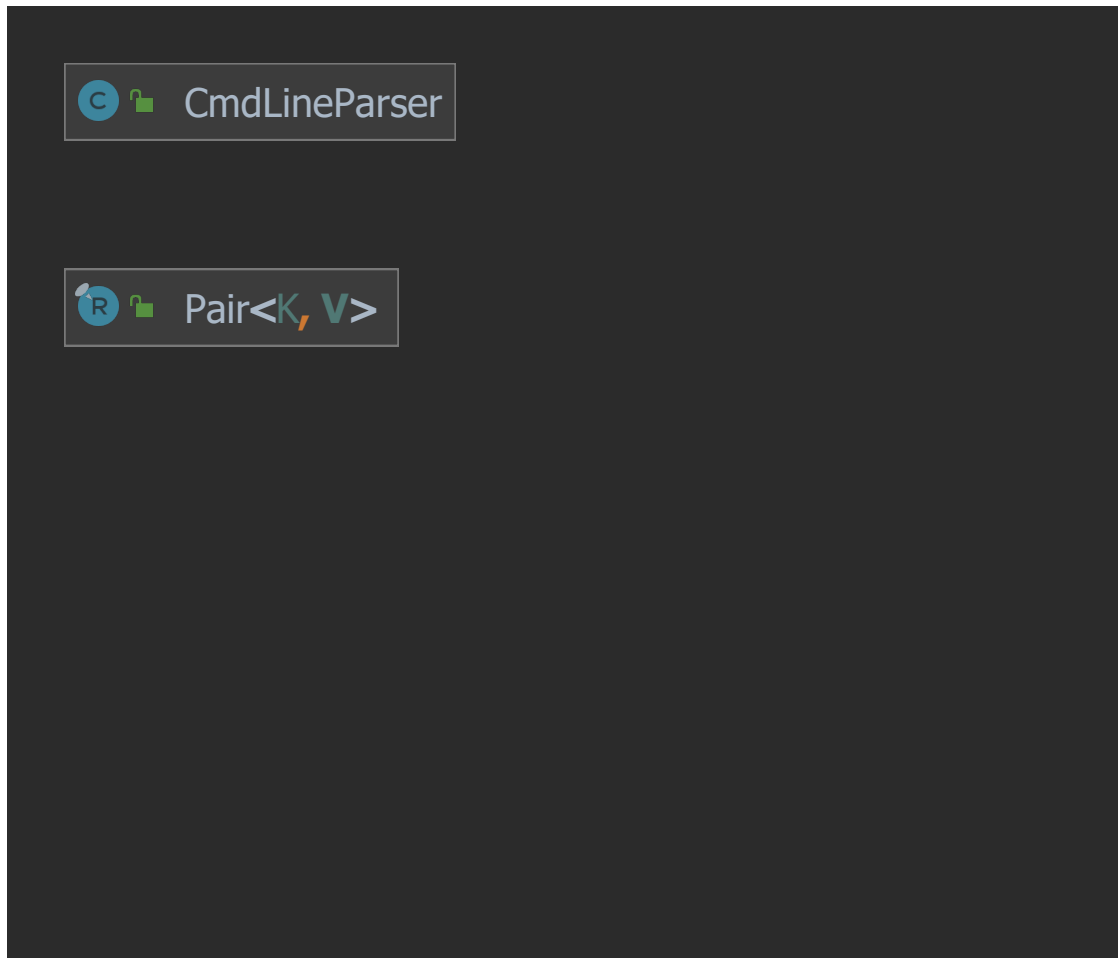


Abbildung 6: utils



A.1.3 Klassen im Paket „framework“

Abbildung 7: Observable





		Observable<T>	
		Observable ()	
		observers	List<Observer <T>>
		notifyObservers (T)	void
		registerObserver (Observer <T>)	void

Abbildung 8: Observer

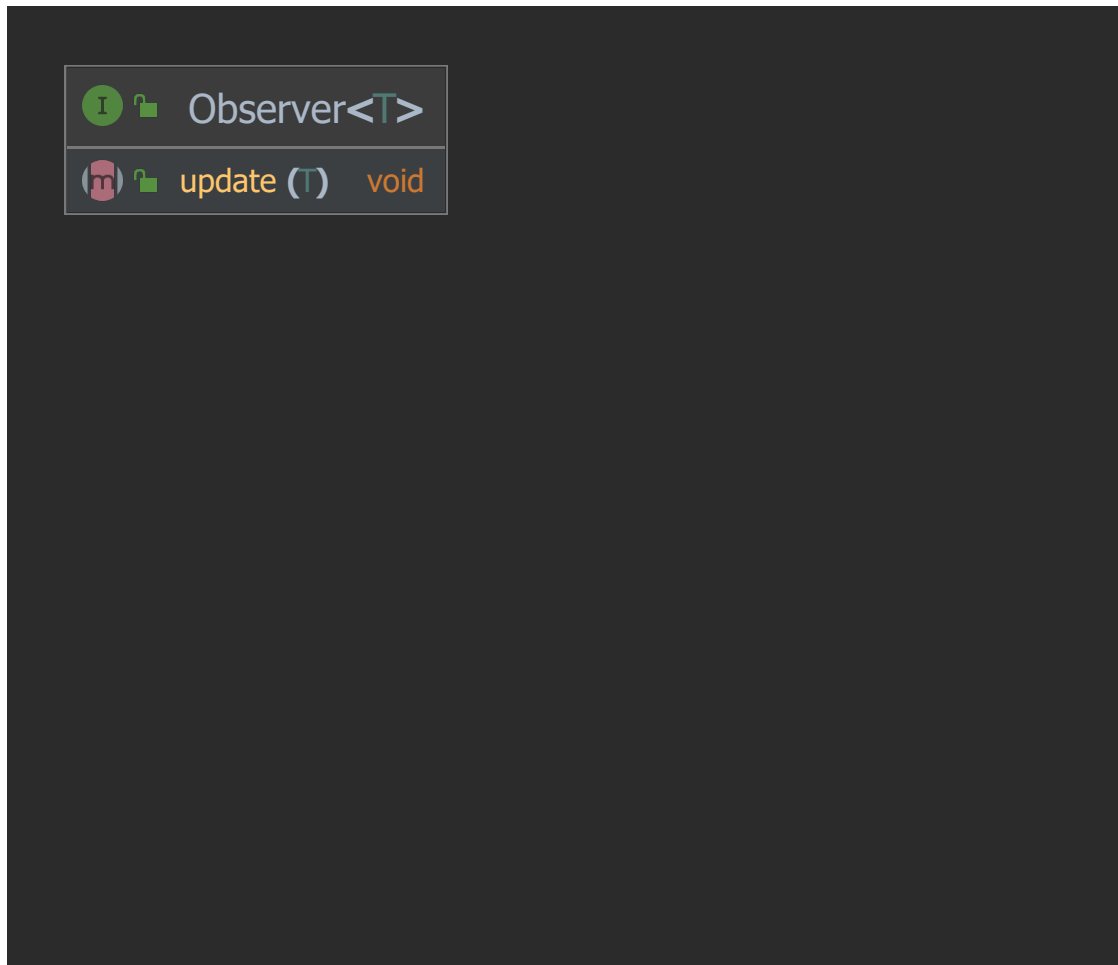


Abbildung 9: ProcessRunnable

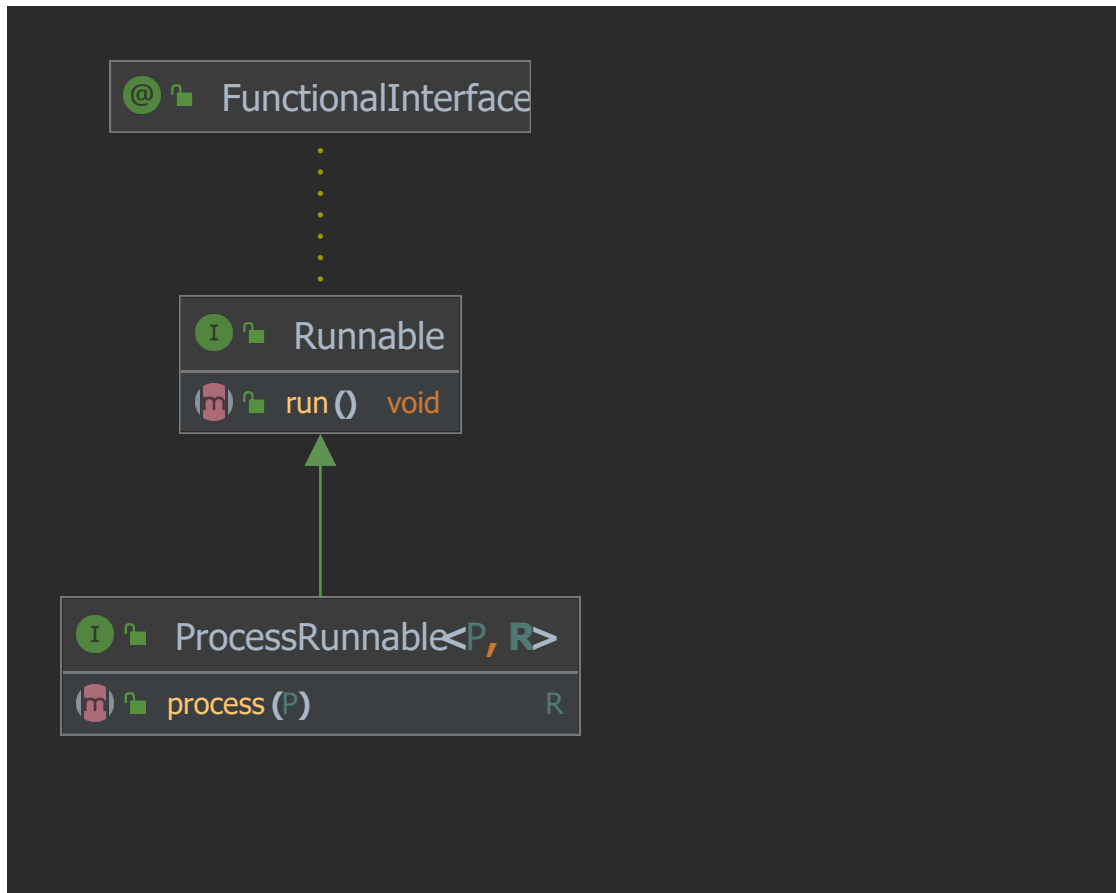


Abbildung 10: ReadRunnable

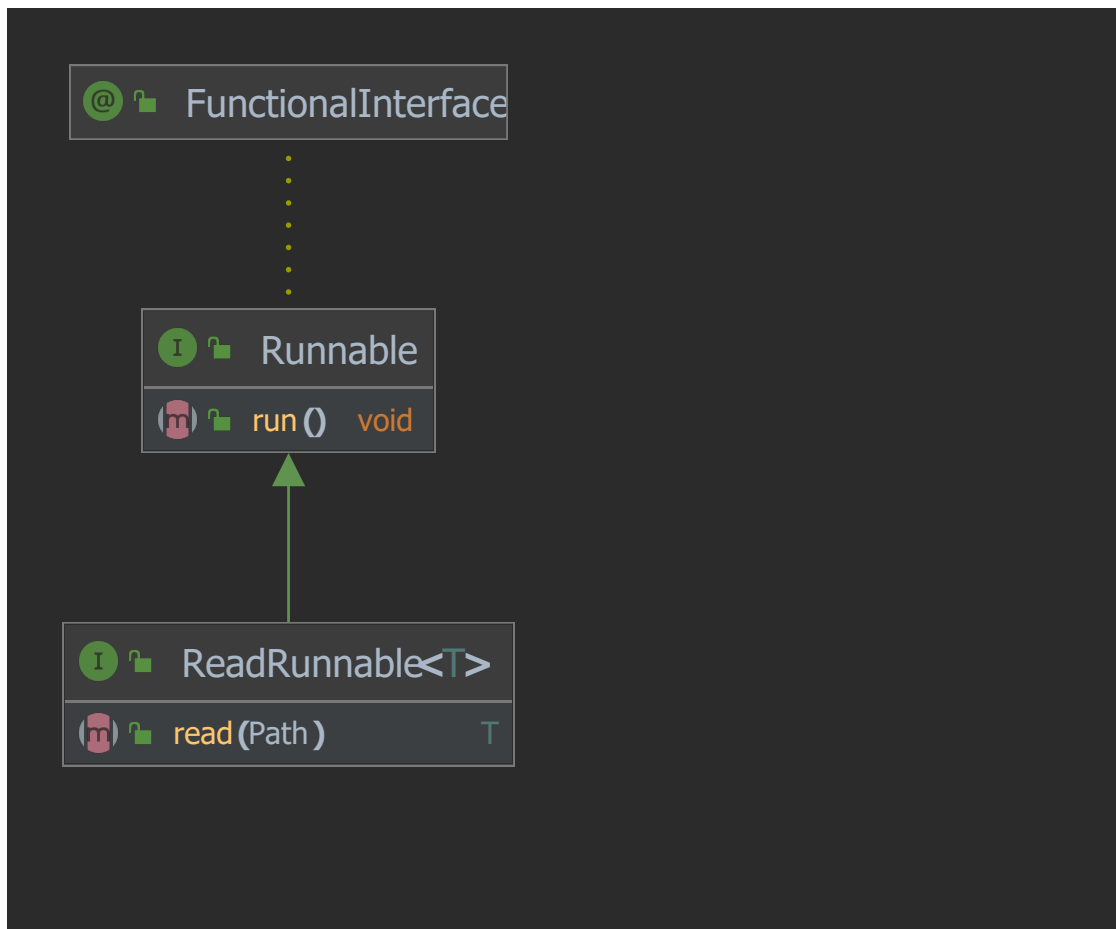
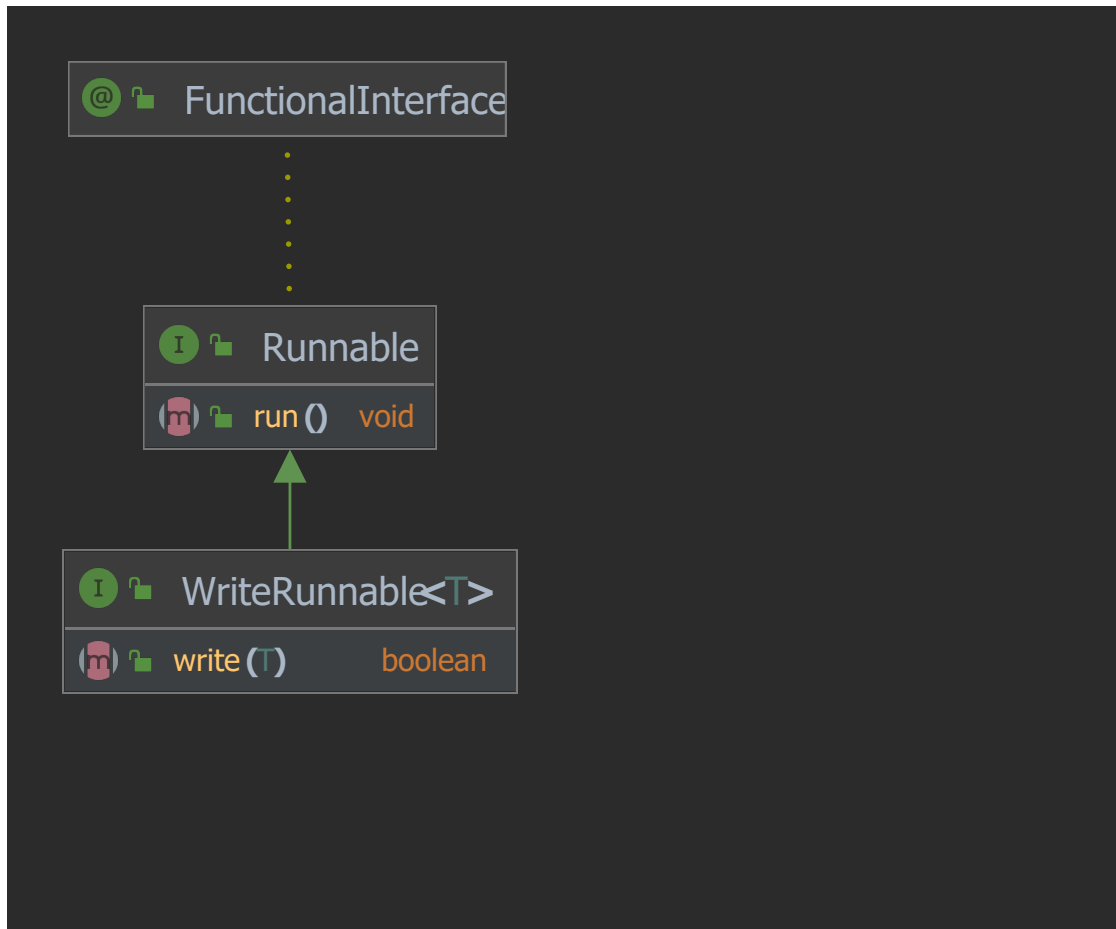


Abbildung 11: WriteRunnable



A.1.4 Klassen im Paket „problem“

Abbildung 12: Algorithms

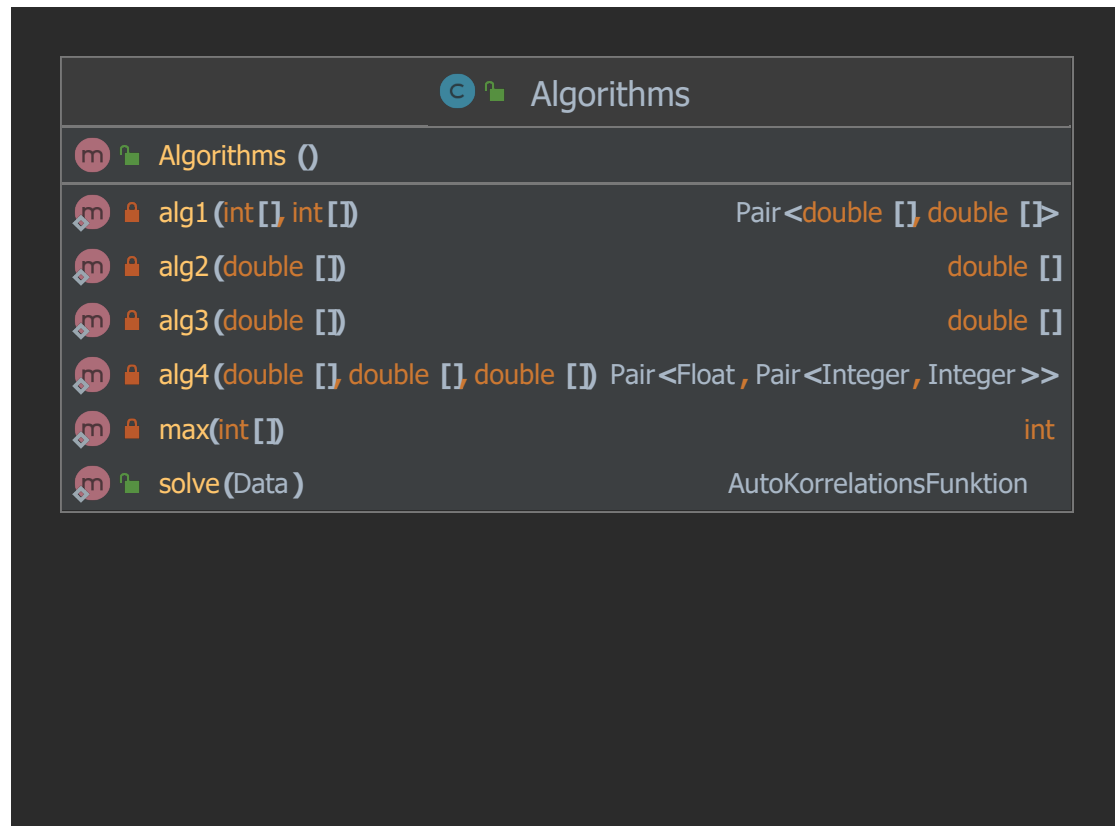


Abbildung 13: AutoKorrelationsFunktion

AutoKorrelationsFunktion		
m	<i>AutoKorrelationsFunktion</i> (String, float, int, int, double[], double[], double[])	
m	<i>fileName()</i>	String
m	<i>fwhm()</i>	float
m	<i>indexL()</i>	int
m	<i>indexR()</i>	int
m	<i>obereEinhuellende()</i>	double []
m	<i>xTransformiert()</i>	double []
m	<i>yNormiert()</i>	double []
p	outputString	String

Abbildung 14: Data











  Data		
 	<i>Data</i>	<i>(String, int[], int[])</i>
 	<i>fileName()</i>	String
 	<i>yStart()</i>	int[]
 	<i>xStart()</i>	int[]

Abbildung 15: ThreadA

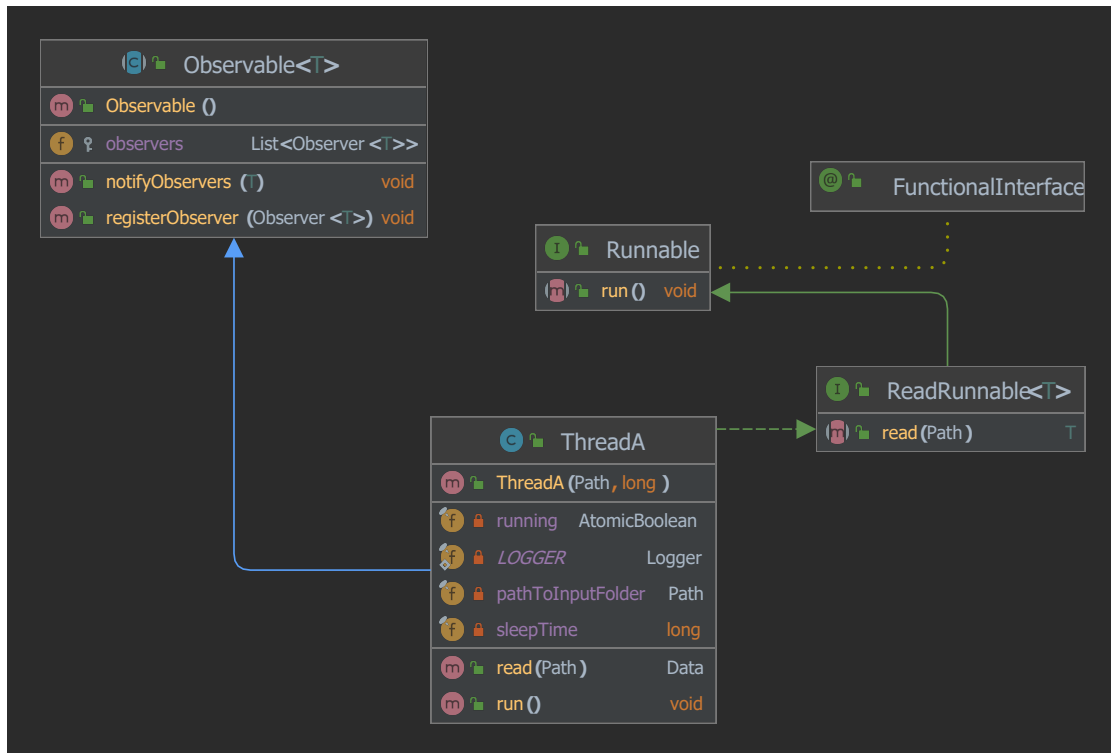


Abbildung 16: ThreadB

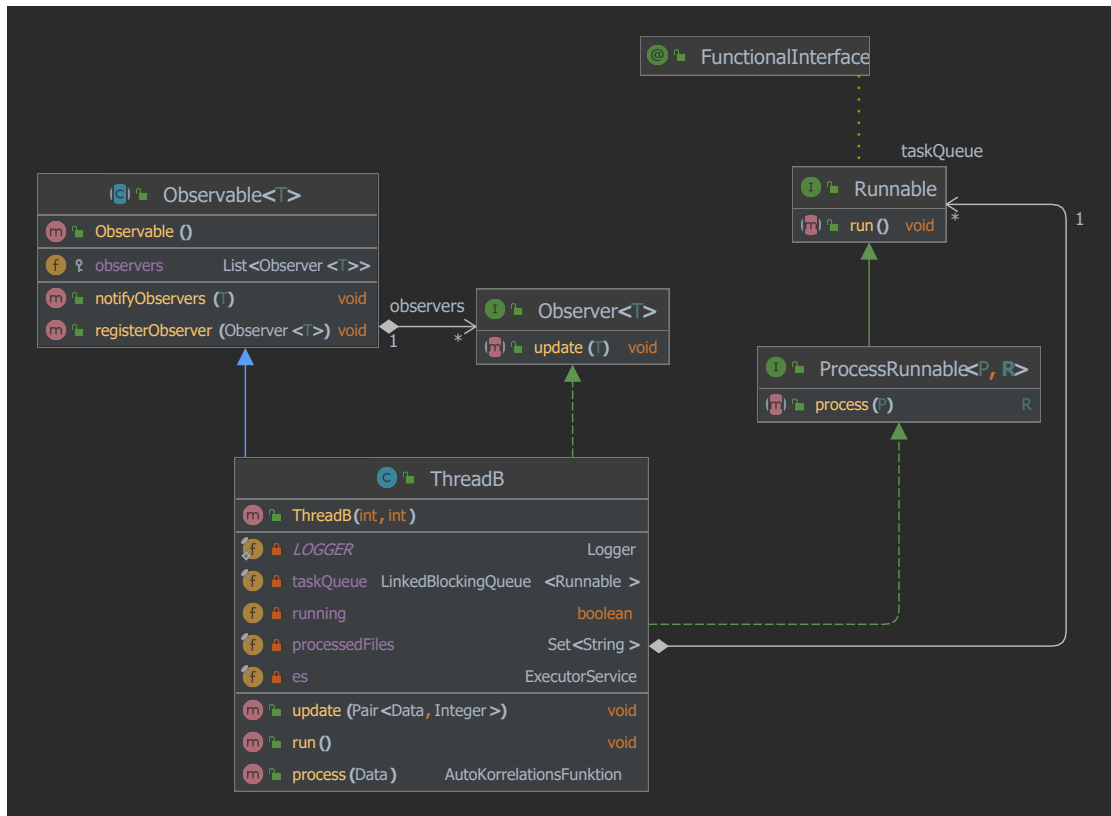
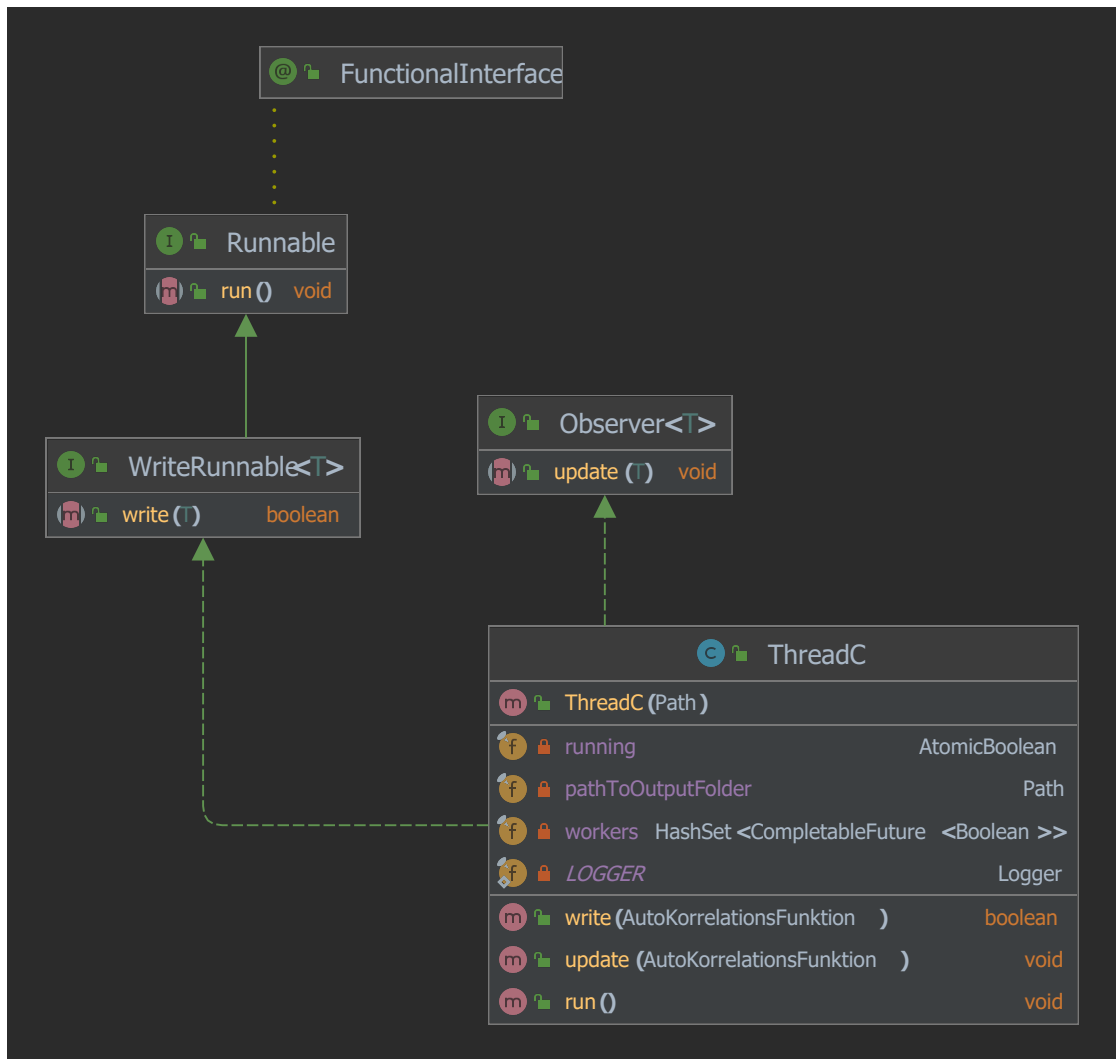


Abbildung 17: ThreadC



A.1.5 Klassen im Paket „problemsimple“

Abbildung 18: ThreadB_simple

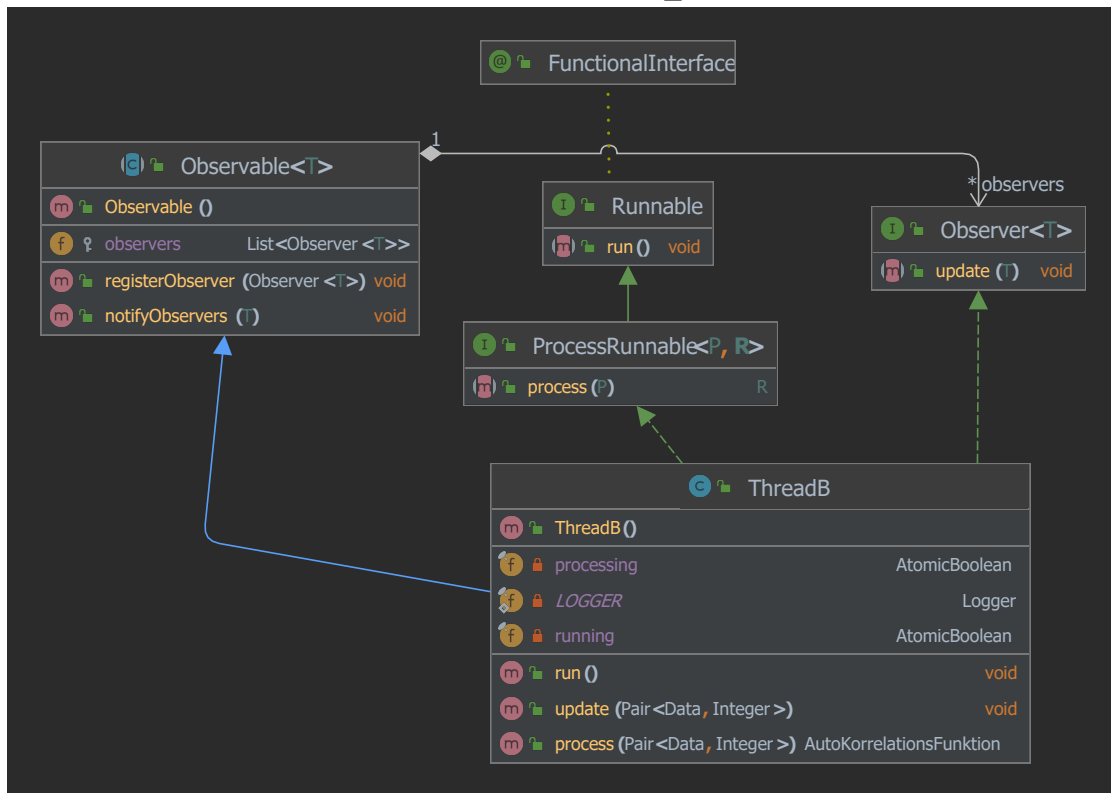
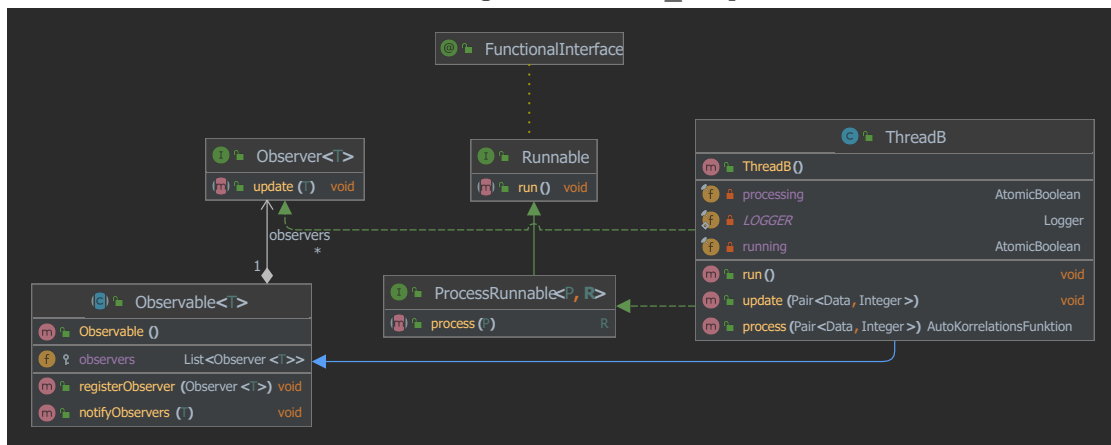


Abbildung 19: ThreadC_simple



A.1.6 Klassen im Paket „utils“

Abbildung 20: CmdLineParser
































































CmdLineParser		
 	CmdLineParser (String [])	
 	outputFolder	String
 	inputFolder	String
  	THREAD_POOL_SIZE_STRING	String
  	INPUT_FOLDER_STRING	String
 	threadPoolSize	int
 	masterWorker	boolean
  	SLEEP_TIME_STRING	String
  	LOG_LEVEL_STRING	String
  	MASTER_WORKER_STRING	String
  	OUTPUT_FOLDER_STRING	String
  	QUEUE_SIZE_STRING	String
  	LOGGER	Logger
  	ROOT_LOGGER	Logger
 	sleepTime	long
 	queueSize	int
  	LOG_STRING	String
 	getThreadPoolSize ()	int
 	getSleepTime ()	long
 	getOutputFolder ()	String
 	getInputFolder ()	String
 	getQueueSize ()	int
 	isMasterWorker ()	boolean

Abbildung 21: Pair

		$\text{Pair}\langle K, V \rangle$	
		$\text{Pair}(K, V)$	
		value	V
		key	K
		$\text{value}()$	V
		$\text{key}()$	K

A.2 Nassi-Shneidermann Diagramme

Abbildung 22: Algorithmus 1

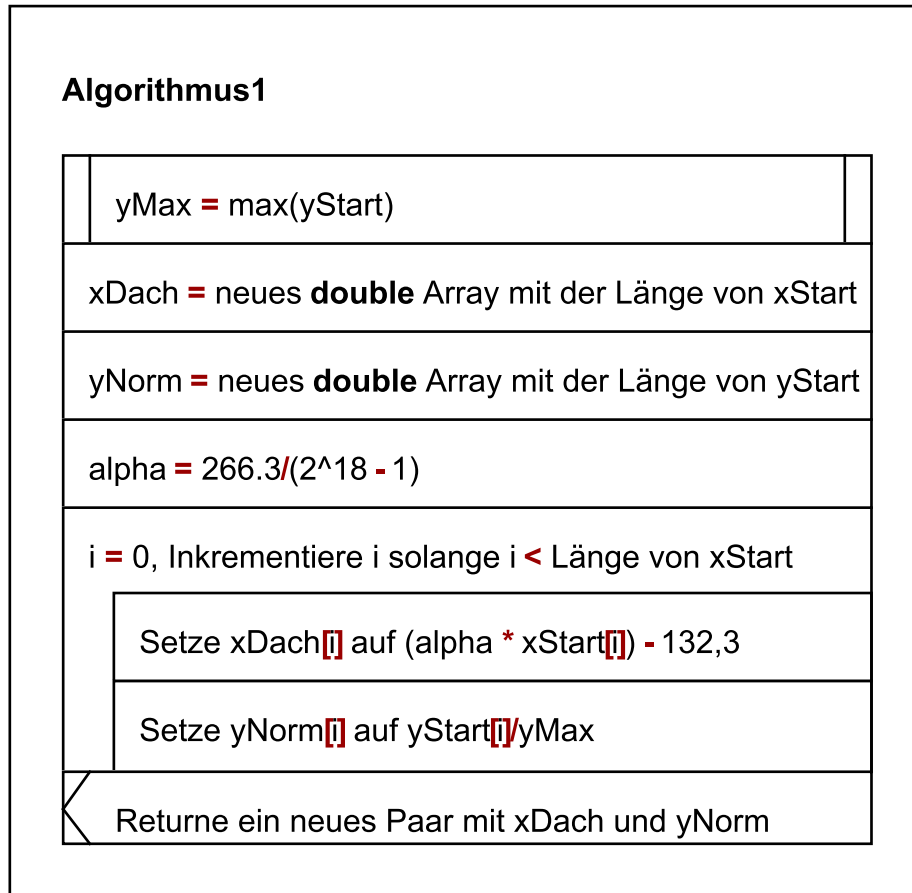
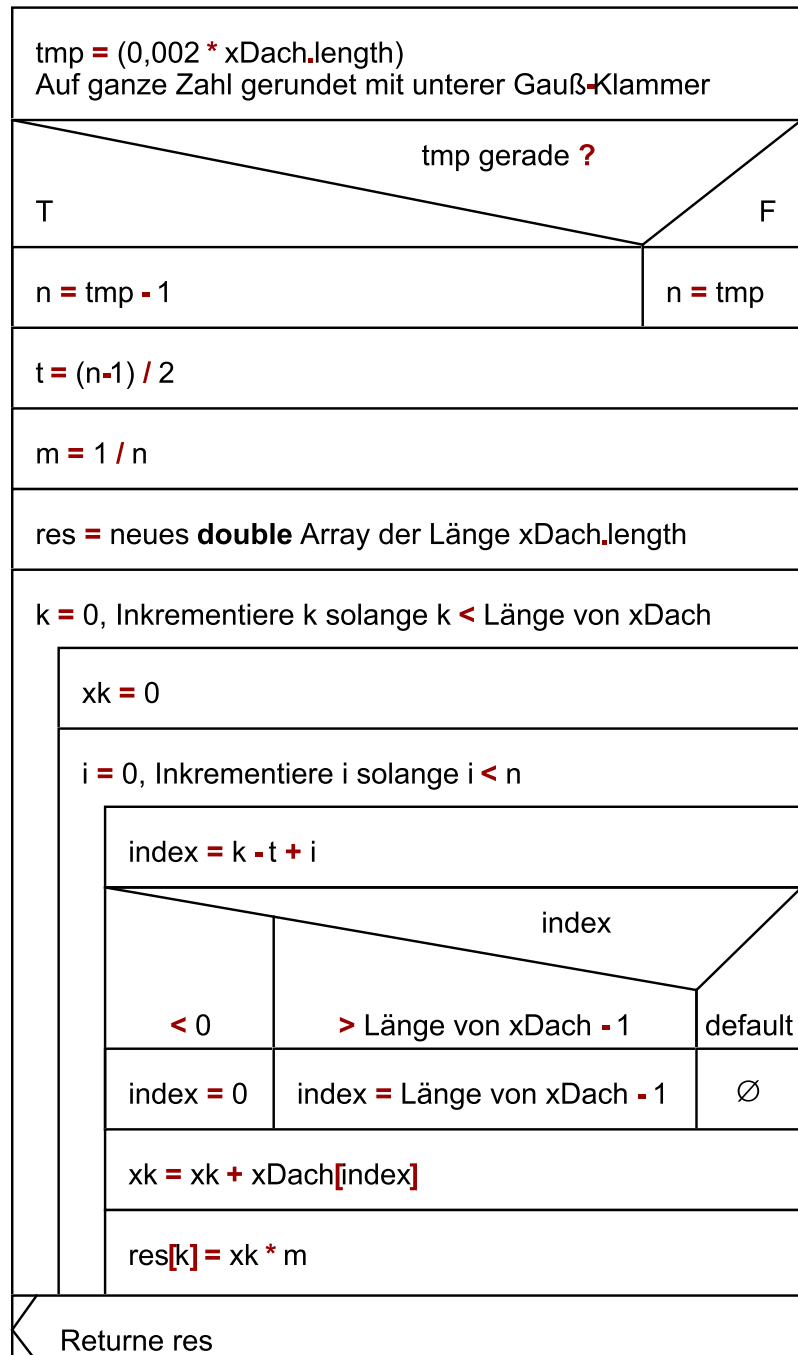


Abbildung 23: Algorithmus 2

Algorithmus2

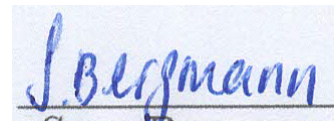


A.3 Sequenzdiagramme

B Eigenhändigkeitserklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Aachen, der 13. Mai 2022
Ort, Datum



Sven Bergmann