

**IHK-Abschlussprüfung Mai 2022**

**Dokumentation zur Entwicklung des Softwaresystems: „AKF-Simulator“**

*Vorgelegt von: Schmidt, Klaus (Matrikelnummer: 3233918)*

*IHK-Ausbildungsnummer: 0000921564*

*Institut:*

*FH Aachen Fachbereich Elektrotechnik und IT-Sicherheit (FB5 Standort  
Eupenerstraße 77)*

## Inhaltsverzeichnis

Aufgabenanalyse .....	4
Sachkontext .....	4
Anforderungen .....	5
Verbale Beschreibung und Diskussion der realisierten Nebenläufigkeit .....	6
Hintergrundinformationen zu Threads und Prozessen .....	6
Datenmodell .....	7
Anforderungen an den Lesethread: .....	7
Grundprobleme „Absprache“ und „Blockade“ bezüglich der Nebenläufigkeit .....	9
Änderungen gegenüber dem handschriftlichen Entwurf vom 09.05.22 .....	12
Strukturbeschreibung des nebenläufigen Systems .....	14
Mathematische Beschreibung der vier Algorithmen der Klasse AKF_Loeser .....	18
1. Umrechnung der x-Werte in Picosekunden .....	18
2. Glättung der Daten .....	19
3. Glättung der Daten .....	20
4. Ermittlung der Pulsbreite .....	20
Nassi-Shneidermann-Diagramme zu den 4 realisierten Algorithmen .....	20
Methode zur Glättung der X-Eingabewerte/ Ermittlung gleitender Mittelwerte .....	21
Methode zur Bestimmung der approximierten Werte zur oberen Einhüllenden .....	22
Benutzerdokumentation .....	23
Endbenutzeranleitung .....	23
Manueller Aufrufbefehl zur Benutzung des Systems mit einer Eingabedatei .....	23
Mögliche Fehlermeldungen gegenüber dem Benutzer .....	24
Format der Eingabedaten .....	25
Format der Ausgabedaten .....	26
Entwickleranleitung .....	26
Systeminformation und Hardware .....	26
Auflistung benutzter Hilfsmittel .....	26
Entwicklungsumgebung .....	26
Visual Paradigm .....	27
Tool zur Erstellung von Nassi-Shneidermann-Diagrammen .....	27
Tool zur Visualisierung der Ausgabedateien .....	27
Information zur „Java-Laufzeitumgebung“ (JRE) und zum „Java-Development-Kit“ (JDK) .....	27
Aufbau der Abgabedatei .....	28
Erstellung einer ausführbaren .jar – Datei .....	28

Liste der benutzbaren Klassen und Schnittstellen, Sequenz und Klassendiagramm des Gesamtsystems .....	29
Ablaufsequenz der main() – Methode als Sequence-Diagramm.....	30
UML2.0 - Klassendiagramm des gesamten Systems .....	31
Erklärung und Veranschaulichung der 3 Run()–Methoden der Threads (Nassi-S. Diagramme) .....	32
1. run()-Methode des Leser-Threads .....	32
2. run()-Methode des Rechner-Threads.....	33
3. run() Methode des Schreiber-Threads .....	34
Zusammenfassung.....	35
Ausblick und Erweiterungsmöglichkeiten .....	36
Quellcode-Listing .....	37

## Aufgabenanalyse

### Kopie der Aufgabenstellung

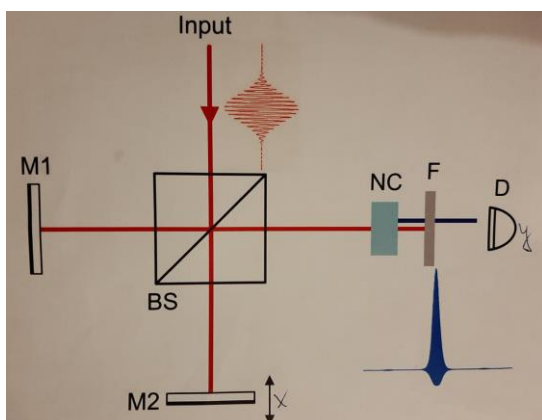
Implementieren Sie eine Simulation des Autokorrelators, in der:

- ein Detektor die Daten aus den gegebenen Dateien ausliest
- die Daten nacheinander die beschriebenen mathematischen Methoden durchlaufen
- die Ergebnisse im gegebenem Format ausgeschrieben werden.

Das Einlesen, Verarbeiten und Ausgeben der Daten muss nebenläufig realisiert werden. Dabei dürfen nur die Standardbibliotheken für Threads und Prozesse der von Ihnen gewählten Programmiersprache verwendet werden. Explizit ausgeschlossen sind externe Bibliotheken wie z.B., aber nicht ausschließlich, OpenMP, MPI, PoenACC, o.Ä., welche eine Nebenläufigkeit bereits implementieren.

### Sachkontext

Die gegebene Aufgabe verlangte eine Simulation eines optischen Autokorrelators, einer Versuchsanlage zur Bestimmung der Selbstähnlichkeit von Signalen, in diesem Fall optische Laserimpulse. Im gegebenen Versuchsaufbau wird ein Laserstrahl über den verschiebbaren Spiegel M2, den halbdurchlässigen Spiegel BS und den Spiegel M1 so an den Detektor D weitergeleitet, dass auf dem Weg dorthin durch Frequenzverdopplung des Lasers in einem Kristall und durch eine Filterung anhand von Änderung der Position des Spiegels M2 im Detektor D die Autokorrelationsfunktion gemessen werden kann.



Die veränderlichen Größen für die verlangte Simulation der AKF-Bestimmung sind die Laufzeit der Lasersignale ausgehend von der veränderlichen Position des Spiegels M2 (Zeitangaben bzw. x-Werte) und die Detektorwerte zur Autokorrelation (Leistungsangaben in Watt, Y-Werte)

### Anforderungen

Anhand eines Datenflusses von Ausgangswertpaaren (X,Y) sollen vier Algorithmen im Zuge der AKF-Simulation die X-Werte „in Picosekunden transformieren, danach glätten“, Y-Werte normieren, der Verlauf einer „Hüllendenkurve“ an den normierten Y-Werte approximiert werden und die Pulsbreite in Bezug auf alle Detekorsignale bestimmt werden. Dies sind die Zielgrößen der Simulation.

Die Simulation sieht vor, einen kontinuierlichen Datenfluss einerseits der Laufzeitwerte des Laserlichtes bei einer Bewegung des Spiegels und andererseits der Autokorrelations-Detektorwerte, somit einen steten Fluss von Wertepaaren, zu simulieren.

Die Datenquelle ist im zu wählenden Datenmodell als leicht austauschbar zu modellieren, d.h. die Quelle der Input-Dateien soll änderbar sein.

Der „Datentransport“, d.h. die Weitergabe von bereits eingelesenen Daten aus den wechselnden Dateien an eine verarbeitende Instanz soll genauer im 50ms-Takt passieren und die unmittelbar zuvor aufbereiteten Daten im „Datenpuffer“ bzw. in einem Zwischenspeicher überschreiben. Neben dem Einlesen bzw. der Darstellung des Datenflusses soll auch die Verarbeitung und auch die Ausgabe der Dateien in einer nebenläufigen Routine passieren. Es bauen die 3 nebenläufigen, d.h. simultan ablaufenden Operationen bzw. Prozesse, erstens Einlesen und Weitergabe aller Datensatz-Datenpaare im 50ms Takt, zweitens Verarbeitung der Daten und drittens die Ausgabe der Daten insofern trotz (!) ihrer Nebenläufigkeit aufeinander auf, als dass sie für ihre jeweilige Aufgabe Datenobjekte aus dem vorherigen Schritt benötigen. Es sind 10 Dateien zur Simulation einer fortlaufend die Rohdaten liefernden Datenquelle vorgegeben worden, welche sukzessive und in einer Dauerschleife als Simulation des permanenten Datenflusses eingelesen werden sollen.

Nebenläufig soll deswegen verarbeitet werden, weil keine Aussage über die Dauer eines einzelnen Prozessdurchgangs (Einlesen, Berechnen oder Ausgeben) getroffen werden kann. Das Ende der Simulation ist dann erreicht, wenn der alle verfügbaren neuartigen Rohdaten verwertet und ausgegeben worden sind. In diesem Fall sollen alle drei Prozesse durch diese Signalinformation zu einem Ende gemeinsamen kommen.

Für die Verarbeitung der Daten ist das mehrfache Einlesen von denselben Dateien nicht mehr relevant, d.h. diese Daten werden nicht erneut in den nebenläufigen Verarbeitungsprozess miteinbezogen, falls Verarbeitungs- und Ausgaberoutine mehr Zeit in Anspruch nehmen als ein vollständiger Durchlauf des Einlesens durch alle Dateien.

Die X- und Y-Wertpaare sind ganzzahlig und positiv in einer Größenordnung im Bereich  $10^4$ , d.h. es liegen im Daten mit dem Typ Integer vor. Aus diesen Basiswerten sind folgende Zielwerte zu erstellen:

### Verbale Beschreibung und Diskussion der realisierten Nebenläufigkeit

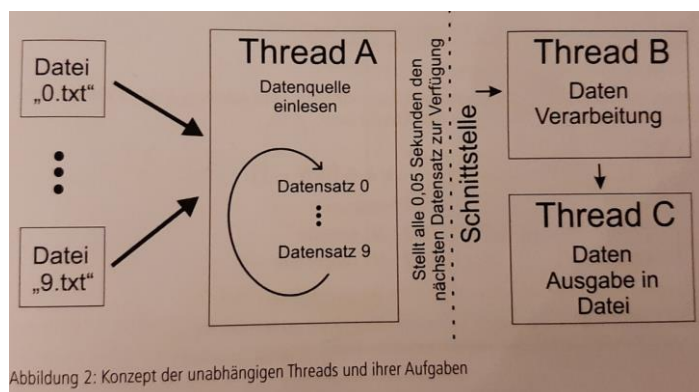
#### Hintergrundinformationen zu Threads und Prozessen

Bei der Diskussion von nebenläufigen Prozessen stellt sich die Frage nach der Realisierung in Prozessen, d.h. Programme und Routinen, welche vom Betriebssystem eigene Betriebsmittel zur Verfügung gestellt bekommen und verwalten oder der Realisierung in Threads, welche stets an einen Prozess und dessen Betriebsmittel (verfügbarer Speicherbereich, „Namespace“, Adressraum) gebunden sind. Ein Thread besitzt kein eigenes Speichermanagement und Speicherschutzmechanismus im Gegensatz zu einem Prozess, sondern stellt nur eine separate Stapelverarbeitungsroutine innerhalb des beherbergenden Prozesses dar. Daher teilt dieser sich mit möglichen anderen Threads des Prozesses dessen Betriebsmittel. Dies kann zu Problemen führen, sollten gemeinsame Ressourcen bei (nahezu) „gleichzeitigem“ Zugriff verändert werden. Überschrieben Änderungen führen in sog. „Race Conditions“ zu Fehlern, rufen Datenverlust hervor oder die Stapelverarbeitung kann „Deadlocks“ dauerhaft zum Erliegen kommen, sollten Informationen zum Bearbeitungszustand verschiedener, voneinander abhängiger Thread nicht korrekt ausgetauscht werden. Der Vorteil von Threads ist hingegen ein schneller und unkomplizierter Austausch von Daten untereinander innerhalb des

Speichermanagements des „umgebenden“ Prozesses, da ein Datenaustausch bei Prozessen hingegen auf zeitintensive „Interrupts“ durch das Betriebssystem angewiesen ist. Weiterhin sind Threads bar eines eigenen Speichermanagements „leichtgewichtiger“ in der Instanziierung als Prozesse. Es können Threads ebenso wie Prozesse die verschiedenen CPU-Kerne heutiger „Multi-Kern-Systeme“ für ihre Aufgabe effektiv parallel nutzen. Ich habe mich für die Nutzung von Threads entschieden, da sie in der Programmiersprache Java leichter zu verwalten sind als Prozesse und den genannten Geschwindigkeitsvorteil mitbringen.

### Datenmodell

In meinem relativ intuitiven Datenmodell habe ich mich stark an der Skizze auf dem Aufgabenblatt orientiert und ein nebenläufiges Design mit drei Threads in der Programmiersprache Java realisiert. Die erste Stapelverarbeitungsroutine („Leserthread“) ist zuständig für das Einlesen und Bereitstellen der Daten in einer bestimmten zeitlichen Taktung. Ein zweiter Thread („Rechnerthread“) dient der sukzessiven Datenverarbeitung von bereits zwischengespeicherten Eingabedaten und der letzte zum Schreiben der Ausgabedateien („Schreiberthread“), wenn Ergebnisdatensätze bereits vorliegen:



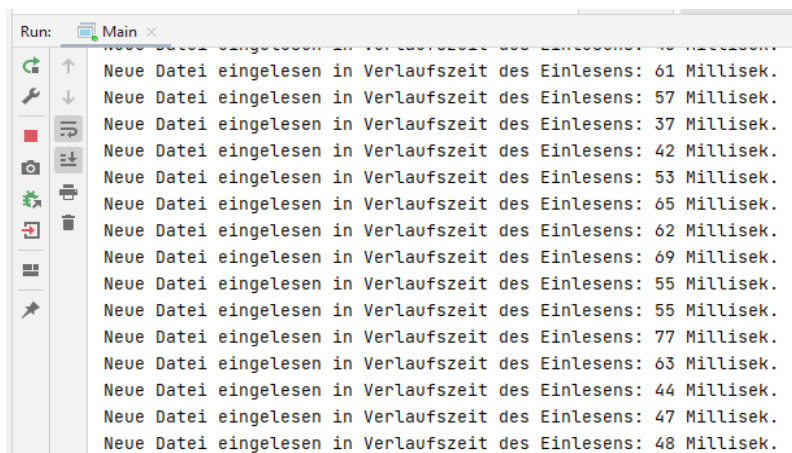
Kernaspekt der Nebenläufigkeit ist in diesem Sachzusammenhang, dass während des Einlesens bzw. der Bereitstellung von Daten bereits – obschon weiteres Einlesen noch passiert – mit diesen bereits gearbeitet wird, danach seitens der Verarbeitung auf die Verfügbarkeit weiterer Daten gewartet wird, währenddessen aber auch schon der verarbeitete Datensatz in Form der o.g. Zielgrößen, ausgegeben werden kann.

### Anforderungen an den Lesethread:

Die Vorgabe eines 20 Hertz-Taktes, mit dem der Lesethread dauerhaft Informationen liefert, ließ sich nur schwer umsetzen, weil die Zeitwerte zum Einlesen der Dateien

schwanken. In meinem System wäre dazu eine weitere gleichrichtende „Zwischeninstanz“ erforderlich gewesen, weil das Einlesen nicht in einem 0.05-Sekunden Takt konstant verläuft, sondern nur um diesen Wert herum schwankt. Es wäre also eine Art „Sammelstelle“ erforderlich gewesen, die nach einer gewissen Zeit genug Rohdaten vorhalten würden, um sie daraufhin in einem konstanten 20-Hertz-Rhythmus weiterzugeben (s. Kapitel Erweiterungen und Ausblick). Da aber die „Einlesezeiten“ nur in geringem Maße um den Wert von 0.05 Sekunden bzw. 50ms schwanken, ist der Gewinn gemessen am Aufwand der Realisierung für mich nicht abzusehen gewesen. Ein leichtes Schwanken der „Einlesezeiten“ um 50ms modelliert meiner Ansicht nach ebenfalls einen relativ gleichmäßigen Zufluss von Informationen.

Insofern alle Dateien auch dieselbe Anzahl von Integer-Datensätzen in ähnlichen Größenordnungen enthielt, kann die Dauer zum Einlesen einer einzelnen Datei durch diese konstante Struktur der Inputdateien des Einlesens keinen stärkeren Schwankungen unterliegen:



```
Run: Main x
Neue Datei eingelesen in Verlaufszeit des Einlesens: 61 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 57 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 37 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 42 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 53 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 65 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 62 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 69 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 55 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 55 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 77 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 63 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 44 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 47 Millisek.
Neue Datei eingelesen in Verlaufszeit des Einlesens: 48 Millisek.
```

Es wurde daher darauf verzichtet, eine weitere nebenläufig konzipierte „Sammelstelle“ mit wirklich konstanter Abgabefunktion von Rohdaten im 50ms – Takt einzurichten.

Weiterhin ist gefordert, dass die Rohdaten fortlaufen während des Einlesens überschrieben werden. Dies geschieht durch einfaches Überschreiben eines Objektattributes „inPutPuffer“ innerhalb des Leser-Threads, wenn die Routine des Einlesens ein neues Input Objekt liefert.



Es ist weiterhin für die AKF- Simulation relevant, dass keine schon einmal registrierten Datensätze ein weiteres Mal in den Verarbeitungsprozess eingehen. Dies entspricht in der Simulation das Ende eines gesamten Bewegungsdurchlaufes des Spiegels M2. In unserer Simulation mit ständig wiederholt eingelesenen Dateien ist diese Situation dann eingetreten, wenn erstmalig eine Datei doppelt eingelesen wird. Die eingelesenen Inhalte sollen dann keinesfalls erneut in den Verarbeitungsprozess eingehen. Demzufolge ist ein entsprechender Kontrollmechanismus zu implementieren.

Eine weitere Anforderung an den Lesethread entsteht aus der Modellierung einer „einfach austauschbaren Datenquelle“. Meinem Verständnis nach bedeutet diese Anforderung neben dem fortlaufenden Einlesen wechselnder Dateien aus einem Ordner auch einfachen zu realisierenden Wechsel des Eingabeordners, was durch die mögliche Angabe des Eingabedatenordners in der Kommandozeile beim Aufruf des Programms flexibel möglich ist.

#### Grundprobleme „Absprache“ und „Blockade“ bezüglich der Nebenläufigkeit

Die Schwierigkeit ergibt sich insbesondere durch die unterschiedlichen Verarbeitungsgeschwindigkeiten der drei Instanzen. In meiner realisierten Implementierung wird an den Berührungspunkten der Threads, d.h. an der Schnittstelle zwischen Lesethread und Verarbeitungsthread sowie beim Verarbeitungs- und Schreibthread ein Zwischenspeicher benutzt, in dem gepufferte Objekte in der Reihenfolge ihres Eintretens in den Datenpuffer zu gegebener Zeit weiterverarbeitet werden können: Zwei Queue Datenstrukturen sorgen nach dem „FIFO“ – Prinzip („First-in-first-out“) für die Pufferung der eingelesenen Daten sowie der auf die Ausgabe „wartenden“, fertig berechneten Daten.

```
public class Controller {  
  
    private final ArrayBlockingQueue<DatenObjekt> queue_gesammelteDetektordaten;  
    private final ArrayBlockingQueue<OutputDatenKollektor> queue_output;
```

Passiert also das Verarbeiten der Daten aus der Signalquelle bzw. in der Simulation aus der Datei schneller als deren Lieferung, so ist die erste erwähnte Queue leer und es müssen Threads „pausieren“ bzw. warten, bis sich die erste Queue wieder gefüllt hat. Problematisch ist, falls nun keine neuartige Detektor-Daten geliefert werden, denn dann würde die Thread-Verarbeitung ewig ruhen, aber nicht beendet werden.

Um häufige Abfrage, ob eine Queue leer ist, zu vermeiden, kommen threadsafe „BlockingQueues“ zum Einsatz. Diese erlauben nur einen Threadzugriff zu einem Zeitpunkt und besitzen eine einstellbare Obergrenze für die zu puffernden Datenobjekte. Versucht ein Thread, mehr Objekte als die maximale Objektanzahl hinzuzufügen, blockiert der Thread solange, bis ein anderer Thread ein Objekt aus der Queue abgerufen hat.

```
/**
 * Initialisiert die leeren Queues.
 */
public Controller() {
    this.queue_gesammelteDetektordaten = new ArrayBlockingQueue<>( capacity: 4);
    this.queue_output = new ArrayBlockingQueue<>( capacity: 10);
    this.idSet = new HashSet<>();
}
```

Dieses blockierende Verhalten steht defacto Widerspruch mit der „Permanenz-Anforderung“ an den Lesethread (s.o.). Versucht dieser Thread, ein eingelesenes neuartiges Input-Objekt eines Datensatzes, also alle X- und W-Wertpaare einer Datei, in die Rohdaten-Queue namens „gesammelte Dektordaten\_queue“ einzufügen, ist jedoch schon aufgrund erreichter Objektobergrenze blockiert worden, muss er auf den Rechnerthread warten, wodurch wiederum ein nicht gewünschtes sequentielles Verarbeiten entsteht. Diesen Widerspruch löst die vorliegende Simulation auf, indem die Obergrenze der ersten Queue zur Objektaufnahme 10 beträgt, also die vorab bekannte maximal mögliche Anzahl aller einzulesender neuartiger Dateien. Durch diese „manuelle“ Festlegung kann das fortlaufende nebenläufige Einleseverhalten des Leserthreads nicht gestört werden. Werden aber Ordner mit mehr als 10 Dateien eingelesen, müsste diese Obergrenze dann auf die entsprechende Dateianzahl erhöht werden.

Meine Klassenkonzeption nutzt eine „Controller“-Klasse als zentrale „Kontrollstelle“, welche beide threadsafe ArrayBlocking-Queues enthält und für alle möglichen Verarbeitungszustände der Threads auch threadsafe „AtomicBoolean“- „Flags“ besitzt, die als Informationsquellen dienen bzw. auf welche die Threads zugreifen können um die o.g. Deadlock-Situation zu vermeiden in Kombination mit einer rekursiven Vorgehensweise der Threads. Es werden so 3 aufeinander aufbauenden

Zustände angezeigt, welche für das Weiterarbeiten der nächsten Prozessstufe jeweils relevant sind:

- Gibt es keine neuen Quellsignale/Dateien mehr? („AlleDateienGeliefert“)
- Sind daraufhin alle „übrigen“ Daten aufbereitet worden?  
(„AlleDatenAufbereitet“)
- Sind daraufhin alle „übrigen“ Dateien ausgegeben worden?  
(„AlleDateienGeschrieben“)

```
public AtomicBoolean alleDateienGeschrieben = new AtomicBoolean( initialValue: false);  
public AtomicBoolean alleDatenAufbereitet = new AtomicBoolean( initialValue: false);  
public AtomicBoolean alleDatenGeliefert = new AtomicBoolean( initialValue: false);
```

While() – Schleifen innerhalb der Thread-Arbeitsmethoden, der Einsatz der „ArrayBlockingQueues“ lassen in Kombination mit diesen drei Flag-Informationen lassen ein echt nebenläufiges Arbeiten der Threads Leser, Rechner und Schreiber in der Simulation zu:

Registriert der Leser, dass alle Quellsignale erhalten worden sind, kann der Rechner-Thread die letzten Objekte in der Signaldaten-Queue entweder noch „leeren“ oder im zweiten Fall nach Ablauf eines Wartezyklus auf neue Daten das Signal setzen, dass alle übrigen Daten nun tatsächlich aufbereitet worden sind. Auf Basis dieser Flag-Information hat der Schreiberthread die Möglichkeit, entweder nach seinen letzten gerade getätigten Dateiausgaben oder nach einem Wartezyklus auf verarbeitete Daten das Abbruch-Flag „AlleDatenAusgegeben“ auf „true“ zu setzen. Dadurch kann er sich somit im nächsten Schleifendurchlauf auch „selber“ zu beenden. Insofern der Leserthread gemäß der Aufgabenstellung permanent Daten einliest, kann er nun auch anhand dieser Information terminieren. Wenn dieser Thread also nicht registriert, dass „AlleDatenGeschreiben“ wurden, liest dieser immer weiter ein. Ansonsten ist die nebenläufige Simulation abgeschlossen. Die Logik der nebenläufigen Prozesse ist dargestellt in einer Auflistung:

1. Der Lesethread liest solange in Schleifen ein, bis alle Dateien geschrieben wurden bzw. das Flag dazu „false“ ist. (1.Flag)
2. Der Rechnerthread rechnet solange oder wartet auf das nächste Datenobjekt, bis alle neuartigen Dateien geliefert wurden, d.h. das zugehörige Flag auch „false“ ist. Danach setzt er das Flag „AlleDatenAufbereitet“ auf true (2.Flag)

3. Der Schreiberthread schreibt solange oder wartet auf das nächste aufbereitete Datenobjekt, bis alle neuartigen Dateien geliefert wurden (1.Flag) UND alle Daten aufbereitet (2.Flag) wurden. Danach setzt er das Flag zum Abbruch „AlleDatenGeschrieben“ (3.Flag) auf true.

Entscheidend für das Auslaufen des Programms ist also die Information, ob noch weitere neuartige Dateien folgen werden. Die Logik zum Erhalt dieser Information ist im Controller-Package (siehe unten) im bereits erwähnten Kontrollmechanismus implementiert.

Wichtige Anmerkung: Leider ist in meiner Implementierung genau diese letzte Informationsübergabe zum Problem geworden, denn der Leserthread erhält bei manchen Durchläufen durch die Versuchsreihen genau diese letzte Information nicht und läuft in einer endlos Schleife weiter, obwohl Rechner- und Schreiberthreads wie gewünscht arbeiten und terminieren. Die geforderten Ausgabedateien werden also erstellt. Trotz viel Aufwand zum Debuggen liegt das Problem vermutlich in einem Datenverlust, der das Flag „alleDatenAufbereitet“ betrifft.

#### [Änderungen gegenüber dem handschriftlichen Entwurf vom 09.05.22](#)

Es hat sehr viele Änderungen gegeben, weniger in Bezug auf nötige Datenhaltungsklassen, sondern hauptsächlich bezüglich der nebenläufig konzipierten Arbeitsweise des Programms. Die im ersten Konzept dargestellte Arbeitsweise ist nicht funktionstüchtig und beruht auf Missverständnissen bezüglich der Aufgabenstellung.

1. Das Einlesen übernahm in meinem ersten Konzept der Main-Thread(). Dies ist bereits ein im Ansatz nicht im Sinne der Aufgabenstellung günstige Überlegung gewesen: Wäre der Main-Thread permanent von Beginn des Programm an mit dem Einlesen neuer Dateien beschäftigt, könnte nicht in simultaner Weise eine weitere nebenläufige Stapelverarbeitungsroutine zum Berechnen der erfassten Werte oder starten. Es ist aber gefragt, dass von Beginn der Simulation an 3 Prozessbearbeitungen simultan stattfinden. Selbst wenn der Main-Thread mit dem Einlesen vorangehen würde, ist mir kein Mittel bekannt, wie dieser durch einen weiteren Thread, z.B. den Schreiber-Thread nach dem Schreiben aller Dateien zuerst gestoppt werden könnte und

schließlich danach die Synchronisation der anderen Stapelverarbeitungen bzw. Threads übernehmen könnte. Daher initialisiert und startet der Main-Thread in meinem jetzigen Konzept „lediglich“ die 3 nebenläufigen Routinen und synchronisiert diese nach Beendigung ihrer Verantwortlichkeit (im Optimalfall..)

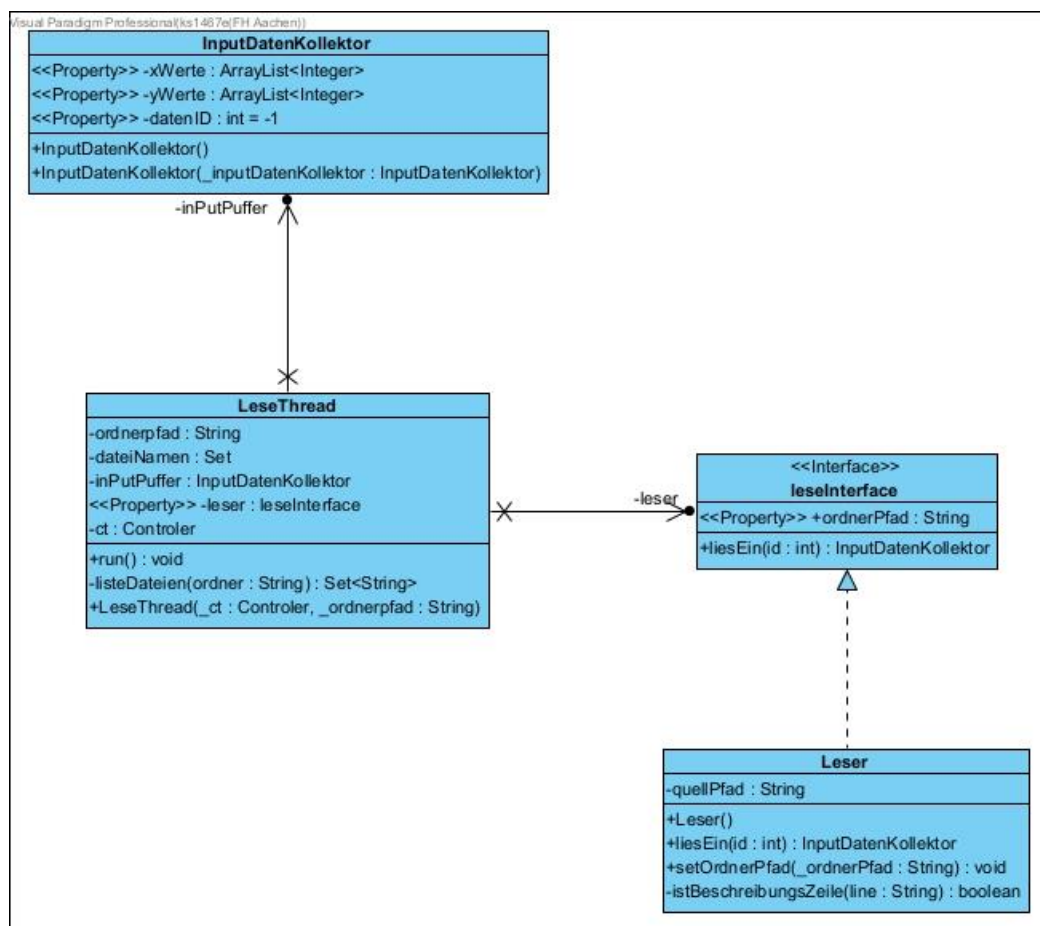
2. Für jede Datei im Main-Thread eine eigene Leser-Instanz einzusetzen und diese alle in einer Liste als Attribut der Controller-Instanz zu sammeln ist nicht sinnvoll, da somit neben Implementierungsprobleme auch Probleme bei den nach Verantwortlichkeit strukturierten Packages „Kontrollieren“ vs „Lesen“ entstehen. Daher habe ich nun eine Klassenstruktur bzw. Packages mit eindeutigen Verantwortlichkeiten erstellt.
3. 10 Leser zu instanziiieren, welche jeweils auf einem eigenen Dateipfad arbeiten und im 50Hz-Rhythmus in einem (Main-)Thread in Schleifendurchläufen einlesen und dann pausieren sind ebenso im Sinne der Aufgabenstellung abwegig. Daher bin ich zum Konzept einer einzigen nebenläufigen Leseinstanz übergegangen, welche aber ständig mittels eines Path-Iterators ihren Dateipfad ändern kann.
4. Im handschriftlichen Konzept ist auch die Rede von 10 parallel gestarteten Verarbeitungs- bzw. Rechnerthreadinstanzen, die jeweils zuständig für eine Datei abwechselnd nach 0.05 Sekunden arbeiten. Um die Aufgabenstellung nicht weiter zu verkomplizieren nutze ich in meinem Softwaresystem nun einen einzigen permanent nebenläufig arbeitenden Rechner-Thread.
5. Im Entwurf sollte ein Boolean-Array im Controller anzeigen, welche Datei bzw. welcher Datensatz aktuell eingelesen wird, weiterhin sollten Flaggs im OutputdatenKollektor anzeigen, für welche der 10 (!) Schreiberinstanzen der gerade berechnete Datensatz bestimmt ist. Insofern diesem Ansatz offensichtlich das Verständnis für nebenläufiges Arbeiten abgeht, da es bei nicht-threadsafes Datenstrukturen zu Race-Conditions kommen kann, werden in meiner jetzigen Fassung thread-safe- AtomicIntegers genutzt.
6. In meinem Entwurf wurde angedacht über gesetzte Flaggs und Size() - Abfragen eine Situation zu vermeiden, in der ein Inhalt im OutDatenKollektor noch nicht gesetzt wurde, obwohl der Schreiber-Threads aber schon initialisiert sein könnte. Diese Situation kann aufgrund interner Steuerung auf

Betriebssystemebene auftreten, wird aber nun durch den Einsatz von blockierenden ArrayBlockingQueues verhindert: Sollte keine abrufbares Datenobjekt vorliegen, muss ein anfordernder Thread warten bzw. blockiert, bis das erste Objekt von einer anderen Stapelverarbeitung in die Queue abgelegt wurde.

### Strukturbeschreibung des nebenläufigen Systems

Das System ist aufgrund aus der Aufgabenstellung resultierenden nebenläufigen Verantwortlichkeiten in 4 wichtige Pakete (Packages) unterteilt: Eingabe, Algorithmus, Ausgabe und Controller. Diese werden im Folgenden mit einem UML-Klassendiagramm und verbal beschrieben.

#### Eingabe-Package

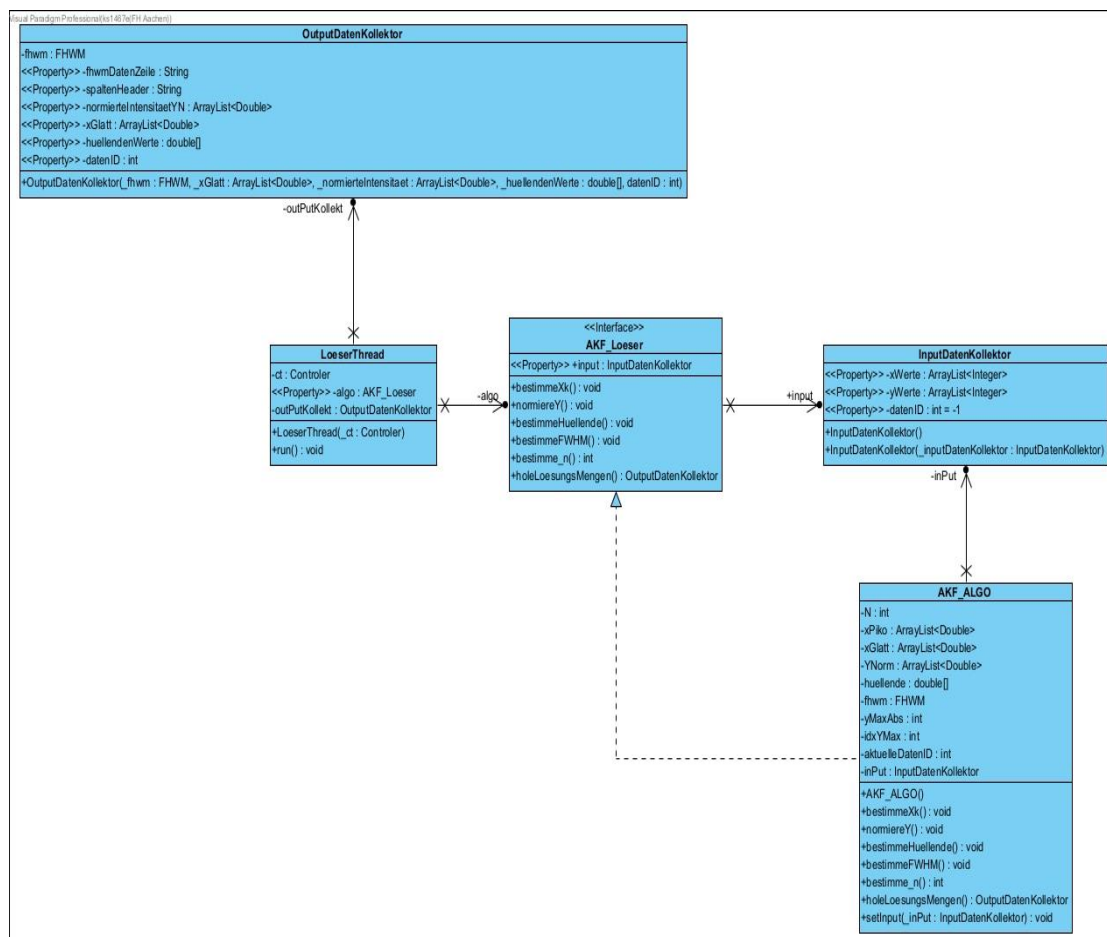


Das Eingabepaket trägt Verantwortung für das permanente nebenläufige Einlesen von Dateien, die damit verbundene Speicherung und Aufbereitung der enthaltenen Rohdaten in „Inputdatenkollektor“-Instanzen und die unmittelbare Zur-Verfügung-Stellung („Abtransport“) von aufbereiteten Rohdaten für den Rechnerthread, bevor

neue Rohdaten generiert werden. Insofern ein kontinuierliches Überschreiben von Rohdaten gefordert ist, wird der „InputdatenKollektor“ als Attribut „inPutPuffer“ des Lesethread angelegt und ständig im Rhythmus des Einlesens überschrieben.

Damit das „Einleseverhalten“ erweiterbar implementiert werden kann, kommt ein Strategy-Pattern zum Einsatz. Wird in weiteren Anwendungsfällen die Form der Eingabedaten eventuell geändert, kann der Anwender durch eine eigene passende Implementierung des „schreibInterfaces“ dieses in der Main()-Methode im „Einlesethread“ setzen und zum Paket hinzufügen. Wichtig ist allein die Konstanz eines zurückgegebenen „InputdatenKollektor“-objektes als Resultat der liesEin(int Id)-Methode. Die übergebene „Id“ des Datensatzes muss dabei ebenso wie eingelesene X- und Y-Werte im „InputdatenKollektor“ return-Objekt gesetzt werden.

### Algorithmus-Package

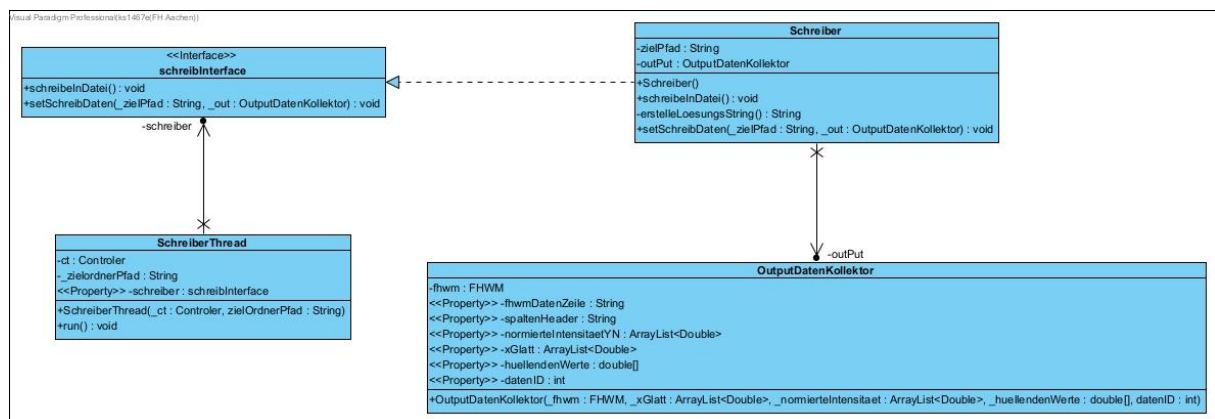


Das Algorithmuspaket übernimmt die Aufgabe der kontinuierlichen Verarbeitung der aufbereiteten Rohdaten durch vier vorgegeben Algorithmen, unmittelbar nachdem diese Rohdaten durch die Datenvermittlung des Eingabepaketes zur Verfügung



gestellt werden. Weiterhin wird dort das Datenmodell der aufbereiteten Daten durch die Klasse „OutputdatenKollektor“ definiert. Diese Instanzen werden nach ihrer „Fertigstellung“ dem Schreiber-Thread im Ausgabepaket zur Verfügung gestellt. Der Algorithmus ist ebenso wie das Leseverhalten offen für Erweiterungen und Verbesserungen implementiert, indem auch hier ein „Strategy-Pattern“ zum Einsatz kommt. Ein anderer Entwickler seine Version des Algorithmus umstandslos in den AKF-Simulator einfügen, wozu er das das AKF-Loeser-Interface implementieren und seine Implementierung vor Start der Simulation in der Main-Methode per Setter-Methode in der LoeserThread-Instanz setzen.

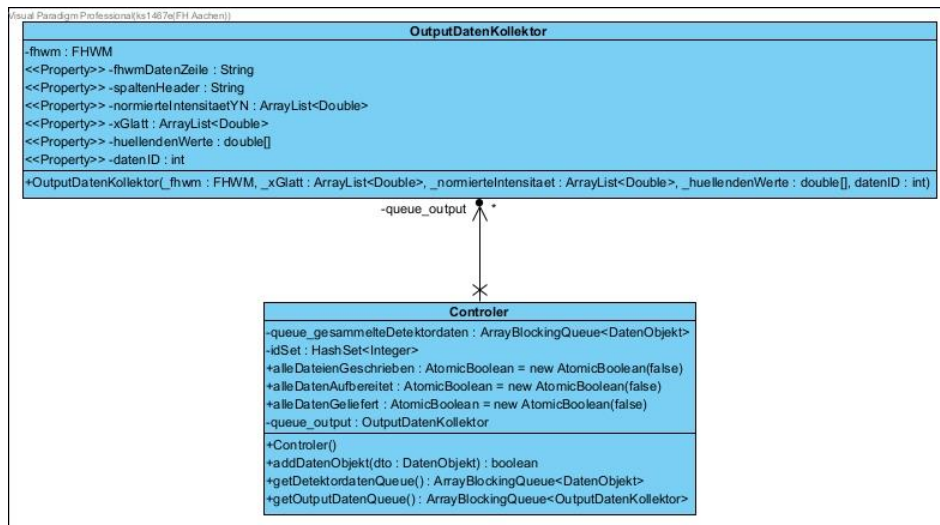
### Ausgabe-Package



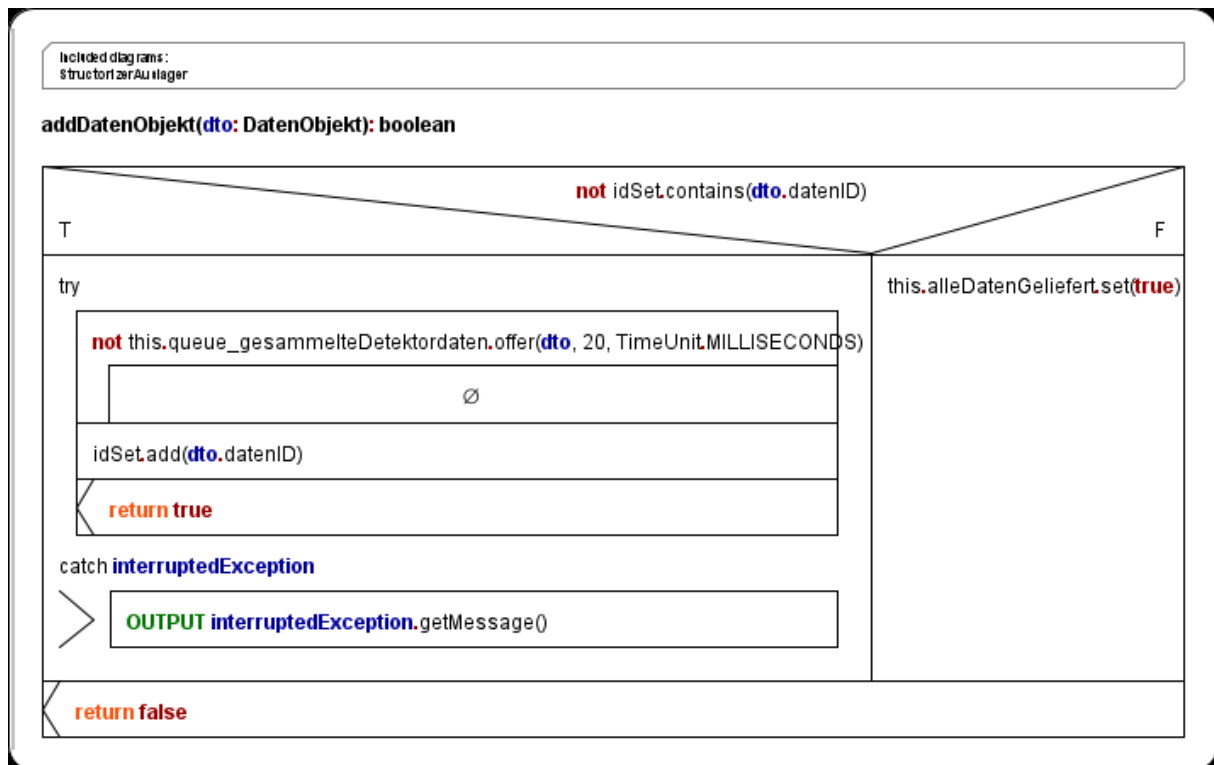
Dem Ausgabepaket fällt allein die Aufgabe des formgerechten Schreibens von vollständig aufbereiteten Daten in Dateien zu sowie und die wichtige Signalgebung zum Ende der Simulation, falls weder neue Daten durch den Leser erwartbar sind noch im System befindliche Rohdaten verarbeitet werden müssen. Um das Schreibverhalten auch erweiterbar für weitere Entwickler zu implementieren, wurde auch hier wiederum ein Strategy-Pattern benutzt. In der Main()-Methode kann so ebenso flexibel das Format der Ausgabedaten durch Hinzufügen und „Setten“ einer weiteren SchreiberInterface-Implementierung zum Package geändert werden.



## Controler-Package



In diesem Paket werden die erwähnten `ArrayBlockingQueues` instanziiert und das entscheidende Signal zum Auslaufen des Programmsystems gesetzt. (Flagg: „AlleDatenGeliefert“). Dies geschieht beim Methodenaufwurf der `addDatenobjekt()` – Methode von Datenobjekten des eingelesenen Inputs zur ersten Queue. Hierbei wird eine `HashSet`-Datenstruktur von Integers genutzt, um im Datenfluss wirklich neuartige Datensätze zu identifizieren. Ein `HashSet` von Integers kann nur unterschiedliche Ganzzahlen beinhalten – es entspricht dem mathematischen Verständnis einer Menge. Für jedes eingelesene Input-Objekt wird deren ID, gegeben im Dateinamen, dort hinzugefügt. Während des Einlesevorgangs und der Weiterleitung der Daten wird über alle Dateinamen im Eingabeordner iteriert. Tritt eine der IDs also zum ersten Mal doppelt auf, dann wurden bereits alle Dateien einmal eingelesen und das Flagg kann gesetzt werden. Weiterhin werden diese Daten dann nicht weitergeleitet an die Detektordaten-Queue. Der Ablauf dieser Methode ist als Nassi-Shneidermann-Diagramm dargestellt:



## Mathematische Beschreibung der vier Algorithmen der Klasse

### AKF\_Loeser

Alle vier Schritte bauen sukzessive auf den Ergebnissen des jeweils vorherigen Schrittes auf (vgl. run()-Methode des Rechnerthread in der Entwicklerdokumentation)

#### 1. Umrechnung der x-Werte in Picosekunden

Diese Transformation der X-Werte geschieht als einfache Multiplikation mit einem sehr kleinen konstantem Faktor  $m = \frac{2663}{10 \cdot (2^{18} - 1)} \approx 3,8147118176e - 6$  und einer

Verschiebung aller Werte um die Konstante -132.3. Somit liegt eine linearer Transformationsvorgang  $x \rightarrow m * x + b$  aller Laufzeitwerte des Lichtes x bei einer Spiegelbewegung an M2 vor, wenn jeder Wert nach Picosekunden transformiert wird. Rechnet man diesen Vorgang anhand einiger X-Startwerte durch, z.B. 127505, so ist festzustellen, dass das Ergebnis der linearen Transformation negative Werte hervorbringt, in diesem konkreten Fall ca. -131,814. In der Physik stellen negative Zeitintervalle aber in Bezug auf eine bestimmten Grundwert 0 eines Experimentes kein Problem dar.

In einem weiteren Teilschritt werden noch die erhaltenen Y-Werte normiert, d.h. durch ihr Maximum dividiert, so dass ihre Werte nun alle in einem Intervall (0..1] liegen, wobei 1.0 dem Maximum dann logischerweise selbst zugeordnet ist.

## 2. Glättung der Daten

Die Glättung der Daten nutzt einen gleitenden Mittelwert in einem bestimmten festen Mittlungsfenster. Dieses mit  $n$  bezeichnet beträgt bei einer Messdatenanzahl  $N$

$$n = \begin{cases} [0.002 \cdot N] - 1 & , \text{für } [0.002 \cdot N] \text{ gerade} \\ [0.002 \cdot N] & , \text{für } [0.002 \cdot N] \text{ ungerade} \end{cases}$$

Damit werden hier pro Mittlungsdurchgang 0.2 Prozent aller  $N$ -Indexwerte in den Vorgang miteinbezogen. Der Wert für das Fenster  $n$  kann durch die Vorschrift nur ungerade sein. Der eigentliche Mittlungsvorgang wird für alle  $k = 0, \dots, N-1$  nun durch die Vorschrift

$$x_k = \frac{1}{n} \sum_{i=0}^{n-1} \hat{x}_{k-\tau+i} \quad \text{mit } \tau = \frac{n-1}{2}$$

ausgedrückt. Das bedeutet anschaulich, dass in einem Array bzw. einer Wertreihe von  $N$ -Zahlen jeweils vom Index 0 an sukzessive aufsteigend eine Zahl „ausgewählt“ wird und nun alle  $n-1$  Zahlen, welche die ausgewählte Zahl „nach links“ und „nach rechts“ umgeben, in den Mittlungsprozess miteinbezogen werden, wobei  $n-1$  wiederum nach der obigen Definition eine gerade Zahl sein muss, da  $n$  stets als ungerade definiert wird. Der Mittelwert über diese „nach links“ und „nach rechts“ umgebenden Zahlen inklusive der ausgewählten Zahl in einer Wertreihe bzw. einem Array von Zahlen wird nun der „Mitte“, also genau der zu Beginn ausgewählten Zahl, als gleitender Mittelwert zugeordnet.

Was passiert aber, wenn nach links und rechts am Rand der gedachten Zahlenreihe „kein Platz“ mehr für das Mittlungsfenster bleibt bzw. keine Zahlen existieren, die für dieses dem Datensatz fix angepasste Fenster noch nötig wären? In diesem Fall wird jeweils ein „einfacher“ Mittelwert diesen Randfall-Elementen zugeordnet. Die Mittlung bezieht sich dann auf ihren Abstand zum Rand bzw. ist stets bezogen auf die Anzahl von vorhandenen Elementen bis zum Rand. Das bedeutet, dass für alle Randfälle das Mittlungsfenster sozusagen erst „langsam“ aufgebaut bzw. „hochgefahren“ wird, bis es den vollen Umfang „ $n$ “ erreicht hat. Der Tau-Wert bezeichnet den konstanten „Ausschlag“ des Mittlungsfensters in der Indexmenge nach links bzw. rechts jeweils betrachtet vom ausgewählten Element, wenn die volle Fenstergröße abseits vom Rand erreicht wurde.

Infolge der Zuordnung eines gleitenden Mittelwertes sind die geglätteten X-Werte nun monoton (steigend).

### 3. Glättung der Daten

Bei dieser Methode wird der Einfachheit halber zwecks Findung einer einfachen „Hüllendenkurve“ eine stufenförmige Treppe über alle normierten Detektor-Messwerte, also die bereits normierten Y-Werte, gelegt. Die Treppe erhöht sich immer, wenn ein neues lokales Maximum auftritt und nimmt dessen Höhe ein. Dabei sind die „Suchrichtungen“ nach lokalen Maxima einmal vom linken Intervallstart bis zum globalen Maximum und vom rechten Intervallrand bis zum globalen Maximum.

### 4. Ermittlung der Pulsbreite

Zur Ermittlung der Pulsbreite wird zuerst eine „Grundlinie“ auf den normierten Y-Werten bestimmt. Diese ist der Mittelwert der äußersten linken ein Prozent der Messwerte, d.h. also  $N/100$  wird ganzzahlig gerundet und auf den Wertebereich der Indexmenge „links“, also von 0 beginnend angelegt. Über alle zugehörigen Y-Werte dieser Menge wird nun gemittelt und so der Grundlinienwert gebildet. Nun wird der mittlere Y Wert vom Grundlinienwert in Bezug auf das globale Maximum benötigt, d.h. es wird, da das Maximum nach Normierung 1.0 beträgt,  $(1.0 + \text{Grundlinienwert}) \cdot 0.5$  bestimmt. Diesem „Y-Pulsbreite“- Detektorwert entspricht einmal linksseitig und rechtsseitig vom Maximum, vereinfacht funktional ausgedrückt, ein Wertepaar  $(X, Y)$ , wobei der Y-Wert also zweimal diesen Y-Pulsbreitenwert annimmt. Die eigentliche Pulsbreite ist, wie der Name schon suggeriert, dann der absolute Abstand zwischen den X-Werten dieser beiden Wertepaare. Da die Funktion nun monoton ist, kann man also auch  $X_{\text{rechts}} - X_{\text{links}}$  berechnen.

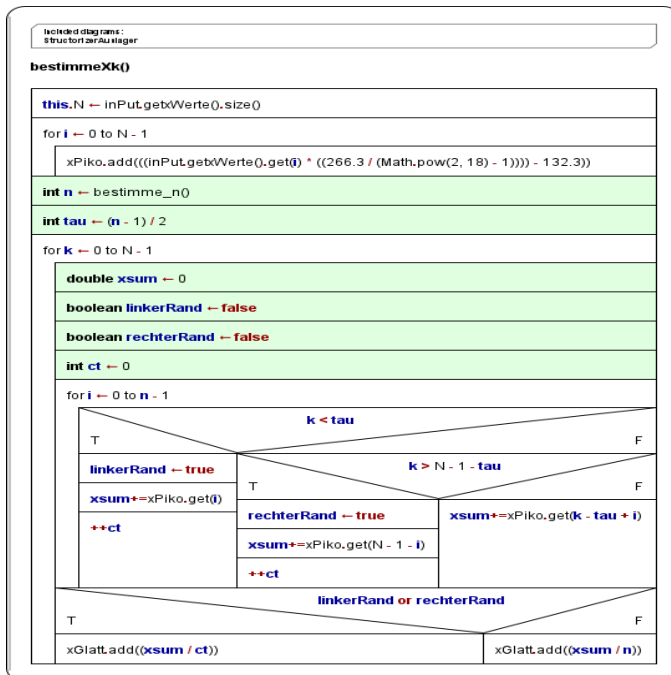
### Nassi-Shneidermann-Diagramme zu den 4 realisierten Algorithmen

In meiner Simulation werden die Zielgrößen der 4 Algorithmen alle als Double-Gleitkommazahlen bestimmt, was den Vorteil erhöhter Präzision gegenüber Rechnungen mit Float-Gleitkommazahlen mitbringt. Die doppelte Bitanzahl von Doubles gegenüber Floats bedeutet auch mehr verfügbare „gleitende“ Nachkommastellen.

## Methode zur Glättung der X-Eingabewerte/ Ermittlung gleitender Mittelwerte

Sie arbeitet auf folgenden Objektattributen der Algorithmusklassse und besitzt aufgrund von zwei verschachtelten Schleifen die Laufzeitkomplexität  $O(n^2)$

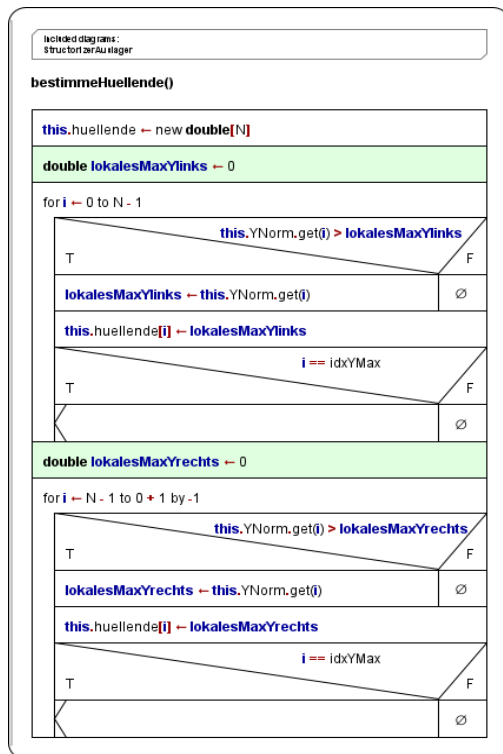
- N: Anzahl aller X-Werte
- xPiko: Eine Array-Liste zur Speicherung aller in Pikosekunden umgewandelter Zeitwerte
- n als ganzzahlige ungerade Größe des Mittelungsfensters (s. Formel oben),
- xsum: Gleitkommazahl zur Zwischenspeicherung von Summenwerten



Methode zur Bestimmung der approximierten Werte zur oberen Einhüllenden

Sie arbeitet auf folgenden Objektattributen der Algorithmusklassse:

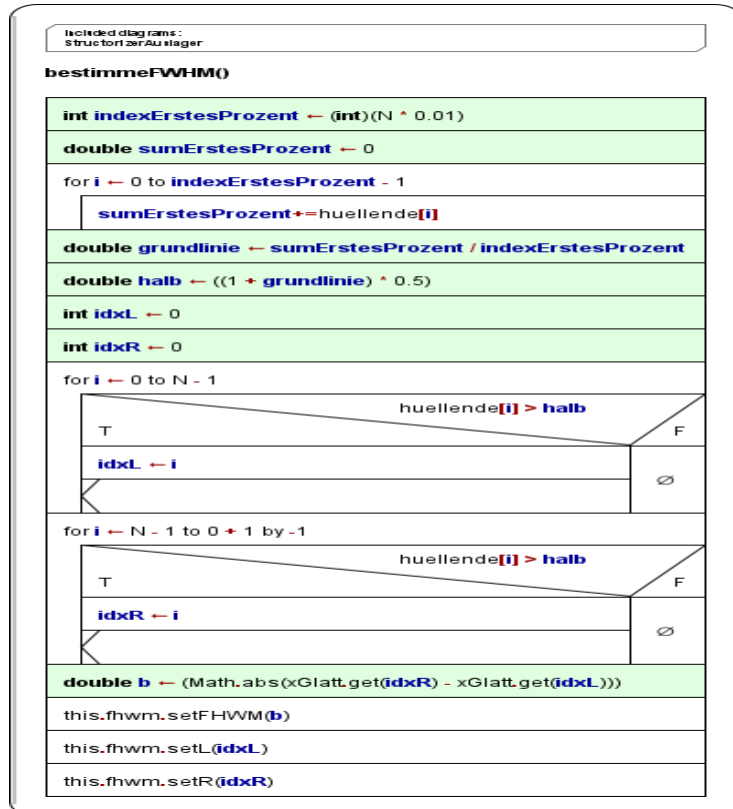
- N: Anzahl aller X-Werte,
- YNorm: Eine ArrayListe zur Speicherung aller normierter Y-Werte
- Integers lokalesMaxYlinks und lokalesMaxYrechts als jeweilige lokale maximale Stufenwerte.
- huellende: Ein double-Array zur Speicherung dieser lokalen Maxima.



## Methode zur Bestimmung der Pulsbreite (FWHM)

Sie arbeitet auf folgenden Objektattributen der Algorithmusklass:

- **huellende**: Das double-Array mit den lokalen Maxima.



## Benutzerdokumentation

Die folgende Benutzerdokumentation ist unterteilt in die Endbenutzeranleitung und die Entwickleranleitung.

### Endbenutzeranleitung

Manueller Aufrufbefehl zur Benutzung des Systems mit einer Eingabedatei

Das Programm „AKF\_Simulator.jar“ kann manuell über ein Programmfenster aufgerufen werden. Der Aufruf per Kommandozeile hat dabei wie folgt auszusehen:

```
java -jar <Programmpfad zum AKF_Simulator.jar> <Eingabedateiordnerpfad>
[<Ausgabedateiordnerpfad>]
```

Dazu gibt man den absoluten Pfad oder vom aktuell in der Konsole angewählten Verzeichnis gesehen relativen Pfad zur „AKF\_Simulator.jar“ Datei als <Programmpfad> ein. Entscheidet man sich für eine relative Pfadangabe und die

Datei liegt in einem Unterordner, dann ist der Bezeichnung des Unterordners ./ voranzustellen.

Der standardmäßige Aufruf des Simulators bei dem mitgelieferten Testdateiordner „data“ würde also lauten:

```
Java -jar AKF-Simulator.jar ./data
```

Die Angabe zum Pfad eines eventuell noch nicht erstellen Ausgabeordners ist optional. Wird keine Angabe getätigt, so werden in den gleichen Ordner der Eingabedateien alle Dateien als „out <Eingabedateiname>.txt“ geschrieben. Sollte ein Ordnername ein Leerzeichen enthalten, dann gilt es zu beachten, diesen Namen in hochgestellte Anführungszeichen zu setzen („<Pfadangabe>“).

Sofern der Ordner Ausgabe noch nicht existiert, wird auch dieser neu erstellt, sofern die Rechte mit Schreib- und Lesezugriff erteilt sind.

Ein Beispielaufruf mit einem relativen Pfad, Leerzeichen im Pfad und der Angabe des Ausgabepfades in einem zu erstellenden Ordner wäre etwa:

```
java -jar AKF_Simulator.jar „./Testbeispiele 1“ ./AusgabeAKF
```

### Mögliche Fehlermeldungen gegenüber dem Benutzer

Anmerkung:

Leider ist das Softwaresystem noch fehlerhaft, es kommt in manchen Fällen zu einer Endlosschleife der Leser-Routine. Ist dies der Fall, so liegen dennoch alle zu den Eingabedaten passenden Ausgabedateien im angegebenen Ordner.

#### *Fehler bezüglich der Eingabedaten und des Eingabedateiordners*

- Der Benutzer gibt einen nicht existenten Ordnerpfad für die Eingabedateien an, so dass eine „EingabeException“ mit folgender Ausgabe geworfen wird:

```
C:\Users\ks1467e\Desktop\GroPro_gg>java -jar AKF_Simulator.jar ./ff
Fehler:.\ff is kein gueltiger Ornder mit Inputdateien! Beende Simulation
```

- Es existieren inkonsistente Daten in einer Datei, z.B. fehlende X oder Y-Werte oder Einträge eines falschen Datentyps. In diesem Fall läuft die Simulation weiter mit denjenigen Dateien, welche vollständig sind, jedoch erhält der Benutzer einen Verweis auf die defekte Datei und die Art des Defektes.



```
C:\Users\ks1467e\Desktop\GroPro_gg>java -jar AKF_Simulator.jar ./data

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.3 = 157 Millisek.

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.8 = 173 Millisek.
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "11.txt"!
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "11.txt"!

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.9 = 149 Millisek.
Schreibe AKF-Werte in die Datei ./data/out3.txt

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.2 = 43 Millisek.
Schreibe AKF-Werte in die Datei ./data/out8.txt
Schreibe AKF-Werte in die Datei ./data/out9.txt
Schreibe AKF-Werte in die Datei ./data/out2.txt

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.7 = 47 Millisek.
Alle daten aufbereitet
Schreibe AKF-Werte in die Datei ./data/out7.txt

.Berechnungen durchgefuehrt in Verlaufszeit (mit Ausgabe): 1406 Millisek.
```

- Bei fehlender Kommentarzeile in der Datei (ohne eine Lehrzeile) erhält der Benutzer nur eine entsprechende Rückmeldung und die Daten werden weiterverarbeitet.
- Alle Datensätze sind inkonsistent, so dass eine „DatensatzInkonsistentException“ geworfen wird mit folgender Ausgabe:

```
C:\Users\ks1467e\Desktop\GroPro_gg>java -jar AKF_Simulator.jar ./data
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "11.txt"!
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "11.txt"!
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "12.txt"!
Es gibt eine inkonsistente Daten-Zeile mit mehr oder weniger als 2 Werten fuer X und Y in der Datei "12.txt"!
Alle Dateien sind inkonsistent. Beende Simulation!
Alle Dateien sind inkonsistent. Beende Simulation!
```

- Der Dateiordner ist vollkommen leer ohne irgendeine Datei:

```
C:\Users\ks1467e\Desktop\GroPro_gg>java -jar AKF_Simulator.jar ./data
Der angegebene Ordner enthaelt keinerlei Dateien. Programmende.
```

### *Fehler bezüglich der optionalen Ausgabeordnererstellung*

- Der angegebenen Ordnerpfad ist ungültig im Dateisystem und es wird eine „AusgabeException“ geworfen:

```
C:\Users\ks1467e\Desktop\GroPro_gg>java -jar AKF_Simulator.jar ./data d:/fc

Verlaufszeit der Berechnung: Fuer den Datensatz Nr.3 = 167 Millisek.
Der die Ausgabedatei enthaltende Ordner im Zielpfad d:/fc konnte nicht erstellt werden. Sie besitzen eventuell keine Schreibrechte
Der die Ausgabedatei enthaltende Ordner im Zielpfad d:/fc konnte nicht erstellt werden. Sie besitzen eventuell keine Schreibrechte
```

### Format der Eingabedaten

Eine Eingabedatei muss einem gewissen Format folgen. Eingeleitet wird sie durch eine Kommentarzeile der Form

**# int pos**

Daraufhin folgen in der nächsten Zeile die Wertepaare Y und X in der Form:

**20114 122188**

**21114 122179**

## Format der Ausgabedaten

Die Ausgabedatei zeigt in ihrer ersten Zeile die Pulsbreite (FWHM) und die Indize der zugehörigen geglätteten X Werte links und rechts vom Maximum an in der Form:

**# FWHM = <fwhm : float>, <indexL : int>, <indexR : int>**

**<x-Werte-geglättet: double> <Y-Werte-normiert: double> <Y-Werte-huellende: double>**

(...)

In konkreter Form hat diese Ausgabe z.B. folgendes Aussehen:

```
# FWHM = 3.7428899400660225e-06, 31602, 33656
# pos int    env
0.0018260094247135369 0.16528505897771953    0.16528505897771953
0.0018260093170075734 0.1652113368283093    0.16528505897771953
0.001826009640125464 0.16444954128440367    0.16528505897771953
...
...
0.0019036737178079002 0.16514580602883355    0.16514580602883355
```

## Entwickleranleitung

### Systeminformation und Hardware

Das Softwaresystem wurde erstellt unter Benutzung eines x64-basierten Windows-10-Betriebssystem „Microsoft Windows 10 Enterprise“, Version 10.0 (Build 19044). Weitere Leistungsdaten meiner Systemumgebung sind 16.0 GB installierter physischer Speicher (RAM), 250GB SSD -Festplatt und einem Intel Core i7-7500U CPU-System mit 2 zu 2,7 GHz getakteten physischen Kernen bzw. 4 logischen Prozessoren.

### Auflistung benutzter Hilfsmittel

#### Entwicklungsumgebung

Die eingesetzte Java-Entwicklungsumgebung ist „IntelliJ IDEA 2021.2.2 (Ultimate Edition)“ der Firma JetBrains, Runtime version „11.0.12+7-b1504.28 amd64“, welche aufgrund des integrierten Code-Ergänzungssystems „Intellisense“ die Fehlererkennung und das Debugging erleichterte.

#### Visual Paradigm

Zur Generierung von UML-Klassen- und Sequenzdiagrammen wurde Visual Paradigm, aktiviert mit einer studentischen Lizenz, benutzt. In diesem umfassenden Tool zur Dokumentation von Softwarearchitekturen wurden diese Diagramme „rückwärts“, d.h. mithilfe der „Reverse Code“- Funktionalität für Klassen und Packages erstellt.

#### Tool zur Erstellung von Nassi-Shneidermann-Diagrammen

Hierzu wurde das kostenlose Tool „Structorizer“ in der Version 3.32-06 benutzt.

#### Tool zur Visualisierung der Ausgabedateien

Hierzu wurde das vom Aufgabensteller zur Verfügung gestellte Programm „FileViewer.py“ benutzt. Dies generiert über den Aufruf

#### **Python FileViewer.py**

die Signalkurven der Autokorrelatorsimulation zu den Ausgabedateien.

#### Information zur „Java-Laufzeitumgebung“ (JRE) und zum „Java-Development-Kit“(JDK)

Java-Quellcode des Softwaresystems wurde in der neuesten Version 18 ausgeführt bzw. geschrieben. Die dazugehörige Java-Standard-Laufzeitumgebung ist „Java Runtime Environment Standard-Edition“ (kurz JRE-SE) zur Ausführung von den in Bytecode kompilierten \*.class Quellcodedateien und liegt in der neuesten Version 18 (build 18+36-2087, Versionsdatum:2022-03-22) vor. Um die Quellcodedateien (\*.java) in Bytecode zu verwandeln wurde das Java-Development-Kit (JDK) mit der entsprechend aktuellen Compilerversion 18 benutzt. Um die aktuell installierten Versionen der Laufzeitumgebung und des Compilers einzusehen, nutzen Sie die folgenden Konsolenbefehle „java -version“ und „javac -version“:

```
C:\Users\ks1467e>java -version
java version "18" 2022-03-22
Java(TM) SE Runtime Environment (build 18+36-2087)
Java HotSpot(TM) 64-Bit Server VM (build 18+36-2087, mixed mode, sharing)

C:\Users\ks1467e>javac -version
javac 18
```

Zu beachten ist durch die Benutzung dieser Compilerversion 18, dass die kompilierten Klassen in der „Class-Version 62.0“ bzw. sogenannter „Major Version 62“ vorliegen und somit von älteren Versionen der Java-Laufzeitumgebung unterhalb

der Version 18 nicht ausgeführt werden können. Jedoch könnte man unter Benutzung entsprechender Befehle kompilierten Code mit älterer „Major Version“ durch eine neuer versionierte „JRE“ ausführen, d.h. die „JRE“ ist abwärtskompatibel.

### Aufbau der Abgabedatei

Das abgegebene „Abgabe.zip“ Archiv besitzt nach dem Entpacken, z.B. unter dem Pfad „Abgabe/“, folgende Inhalte:

- Ordner „/data“, welcher die gegebenen Test und Ausgabedaten enthält.
- die Eigenständigkeitserklärung als „Eigenständigkeitserklärung.pdf“
- eine Windows-Batchdatei „starte\_AKF\_Loeser.bat“ zum Starten des Softwaresystems
- die schriftliche Dokumentation des Softwaresystems als „Dokumentation.pdf“
- das „.jar“-Archiv des Softwaresystems unter „AKF-Simulation.jar“
- den Projektordner inklusive der Ordner der „JavaDoc“ und des gesamten Quellcodes „src“,

### Erstellung einer ausführbaren .jar – Datei

Um den Java-Code der geschriebenen „\*.java“- Dateien zu kompilieren, manövriert man innerhalb der Eingabeaufforderung („cmd“) in den gelieferten „src“-Ordner des Projektes und kompiliert diese Dateien zu „\*.class“- Dateien mit dem Befehl `javac *.java`

**`javac <package1>/<dateiname>.java`**

Sind alle „.java“-Dateien zu „.class“ Dateien kompiliert, wird zur Erstellung eines aufrufbaren .jar-Archivs der Befehl

**`jar -cvfe <Archivname>.jar <Einstiegspunkt> *.class <Package1>/*.class <Package2>/*.class <Package3>/*.class`**

genutzt. Hierbei steht <Archivname> für den gewünschten Namen des „.jar“-Archives, <Einstiegspunkt> für denjenigen Dateinamen, dessen zugehörige „.class“ bzw. „.java“- Datei die das Softwaresystem startende „public static void main(String[] args)-Methode“ ohne Dateitypendung enthält und <Package1> meint etwa Model/\*.class, also den Namen eines Package-Ordners. Danach liegt im „src-Ordner“ des Projektes die gewünschte „.jar-Datei“ vor.

## Liste der benutzbaren Klassen und Schnittstellen, Sequenz und Klassendiagramm des Gesamtsystems

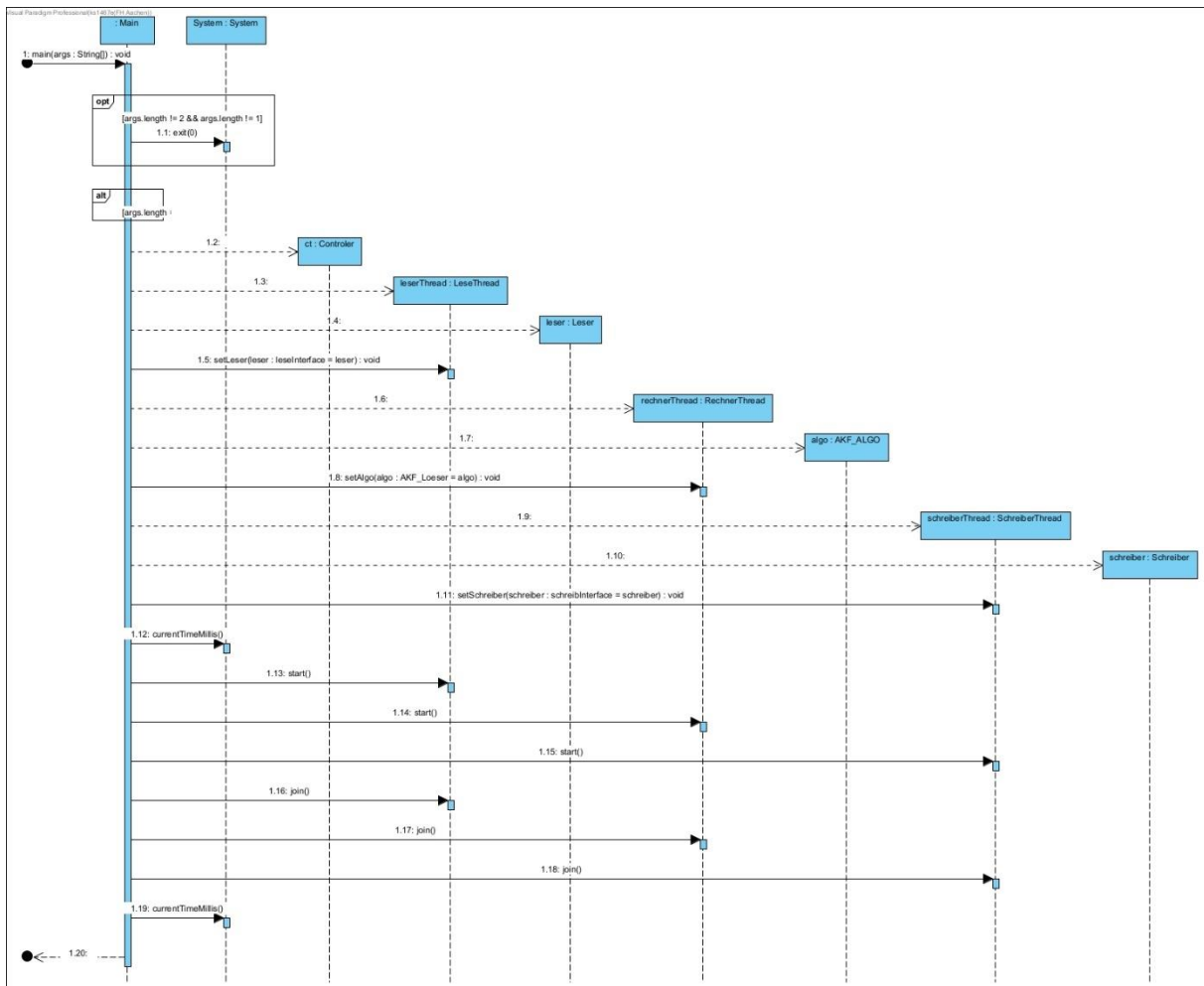
Es folgt eine Übersicht über die benutzbaren Klassen und Interfaces des Softwaresystems, sortiert nach ihrer Package-Zugehörigkeit:

Package	Klassen zur Datenhaltung im Package	Klassen als Interface-Implementierung eines Strategy-Patterns	Basisklassen des Packages zur Ausführung der Nebenläufigkeit	Interface:
Eingabe	Inputdaten-Kollektor	Leser	LeseThread	leseInterface
Ausgabe		Schreiber	SchreiberThread	schreibInterface
Algorithmus	Outputdaten-Kollektor	AKF_ALGO	RechnerThread	AKF_Loeser
Controller			Controler	
Main			Main	
Model	DatenObjekt, FHWM			
Model.Exceptions	Ausgabe_-Exception DatensatzInkonsistent-Exception; DatensatzInkonsistent-Exception; DatensatzInkonsistent-Exception;			

## Ablaufsequenz der main() – Methode als Sequence-Diagram

Zur Veranschaulichung des Programmablaufes zur Instanziierung der zentralen Threading-Klassen dient das folgende Sequence-Diagram. Stichpunktartige Kurzbeschreibung:

1. Prüfung der Kommandozeilenparameter auf Vollständigkeit und den optionalen Ausgabeordner Eintrag
2. Instanziierung von Controller, Leserthread, Leser
3. Setzen des Leser als Realisierung der LeseInterface-Implementierung
4. Instanziierung des RechnerThreads und des AKF\_ALGO
5. Setzen des AKF\_ALGO als konkrete AKF\_Loeser-Implementierung
6. Instanziierung des SchreiberThread und des Schreibers
7. Setzen des Schreibers als konkrete SchreibInterface-Implementierung
8. Starten einer Zeitmessung und Starten aller 3 Thread
9. Zusammenführen der Thread und Beenden der Zeitmessung (Programmende)



```

classDiagram
    class User {
        name
        email
        password
        role
    }
    class Admin {
        name
        email
        password
        role
    }
    class Teacher {
        name
        email
        password
        role
    }
    class Student {
        name
        email
        password
        role
    }
    User <|-- Admin
    User <|-- Teacher
    User <|-- Student
    User --> Admin
    User --> Teacher
    User --> Student
    Admin --> Teacher
    Admin --> Student
    Teacher --> Student
  
```

- 31

## Erklärung und Veranschaulichung der 3 Run()–Methoden der Threads (Nassi-S. Diagramme)

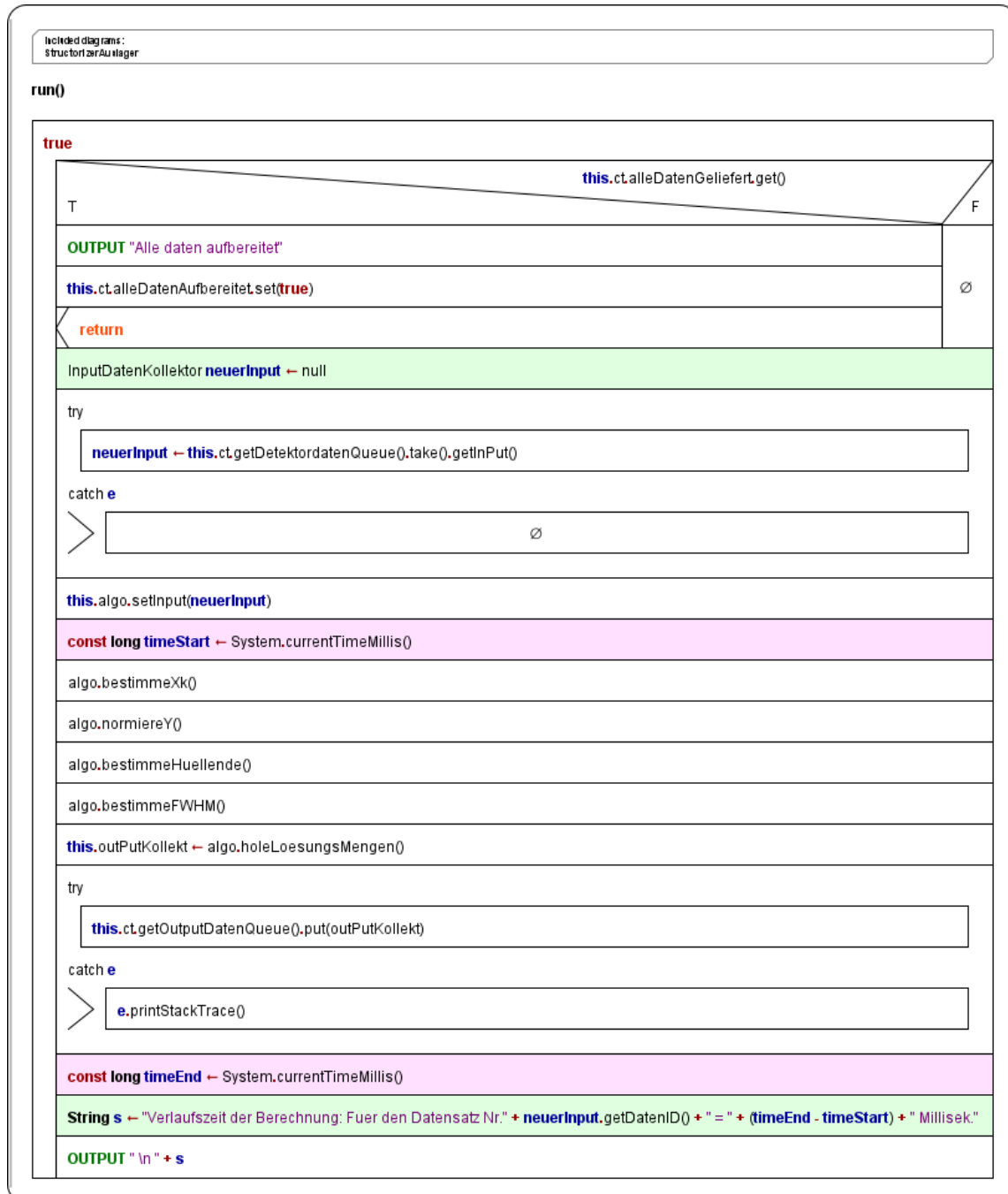
### 1. run()–Methode des Leser-Threads



- Abbruchbedingung für die Routine des Lesers ist das durch den Schreiberthread gesetzt threadsafe Flag „alleDateienGeschrieben“
- Im folgenden „beauftragt“ der Thread seine Attributreferenz Leser, anhand des Dateipfades aus dem PathIterator eine Datei auszulesen und gibt die Anweisung zur Speicherung der Inputdaten in der „Rohdaten-Queue“



## 2. run()-Methode des Rechner-Threads



- Bedingung für den Abbruch der Berechnungsroutine ist zu Beginn nach der while() – Schleife erkennbar: Sind alle Daten seitens des Leserthread geliefert worden, so befindet sich der Rechner im letzten Bearbeitungsdurchlauf durch die Schleife. Infolgedessen kann er das threadsafe Flag-„alleDatenAufbereitet“ als Signal für den SchreiberThread setzen
- Er nach dieser Abfrage seine Daten aus der InputDatenQueue, berechnet die Werte mithilfe seiner Algorithmus-Attributreferenz (s. Strategy-Pattern) und fügt diese der OutputdatenQueue hinzu, sofern der maximale Speicher dort nicht überschritten ist. Sonst blockiert der Thread bis genug Platz frei wird.

### 3. run() Methode des Schreiber-Threads



- In diesem Ablaufschema zu Beginn deutlich, dass der Schreiberthread solange Daten anfordert bzw. erwartet aus der OutPutDaten-Queue und in Dateien schreibt, bis er das Signal vom Leser „alleDatenGeliefert“ in Kombination mit dem Signal vom Schreiber „AlleDatenAufbereitet“ erhält.
- In diesem Fall kann er sicher sein, sich im letzten Durchlauf zu befinden, setzt also „vorausschauend“ das Flag „alleDateienGeschrieben“
- Die Anforderung der Daten aus der OutputDaten-Queue geschieht immer während des Setter-Aufrufes auf dein Schreiber-Attribut (Strategy-Pattern)

## Zusammenfassung

Im vorgelegten Softwaresystem zur Simulation eines AutoKorrelators bzw. von dessen Datenverarbeitung wurde Wert gelegt auf eine einfache Implementierung der Nebenläufigkeit zu drei Verarbeitungsroutinen: Einer permanent datenliefernden Einleseroutine in einem leicht schwankenden Rhythmus um 50ms, einer Verarbeitungsroutine und einer Schreibroutine. Die Nebenläufigkeit wurde in Java über Threadimplementierung umgesetzt. Verantwortlichkeiten wurden in Packages getrennt. Deren Struktur wird bestimmt durch die aus dem Sachzusammenhang nötige Funktionalität des jeweiligen Verarbeitungsvorgangs bzw. des dazugehörigen Datenaustauschs. Die dazu nötigen Datenhaltungsklassen („Containerklassen“) wurden in jedem Package passend angelegt.

Die Controller-Klasse nutzt zwei Array-BlockingQueue-Implementierungen, um einen threadsafes Datenaustausch der nebenläufigen Routinen an den Schnittstellen „Lesen -><-Berechnen“ sowie „Berechnen -><-Ausgeben“ realisieren zu können. Das System ist über die Implementierung des Strategy-Patterns bezüglich des Leseverhaltens, der Durchführung des Algorithmus und des Schreibverhaltens leicht erweiterbar

Leider ergab sich ein für mich nicht mehr zu lösendes „DeadLock“ - Problem, welches nicht in jedem Programmdurchlauf, aber dennoch häufig auftritt: Zum Programmende wird der Leser-Thread nicht ordnungsgemäß beendet, obschon alle Daten erfolgreich ausgeschrieben und zugehörige Berechnungen getätigt wurden, so dass das Programm „hängt“. Es kommt wahrscheinlich zu einem Datenverlust bzw. Signalverlust im Kontext der threadsafes Flags. Ich habe aber von einer Sequenzialisierung des Programms abgesehen, nur um dieses Problem zu umgehen, weil die Nebenläufigkeit ausdrücklich das Kernanliegen des Systems ist.

## Ausblick und Erweiterungsmöglichkeiten

Als Erweiterung des Systems hin zu einer besseren Performance ist der Ausbau zu weiterem „Multithreading“ vorstellbar, um z.B. bei größeren Datensätzen eine zügigere Auswertung zu erhalten. Das festgestellte Verhältnis von Rechenzeit und Zeit zum Einlesen einer Datei betrug ungefähr 100ms zu 50ms, d.h. 2:1, so dass ein zweiter Rechnerthread bei konstanter Rechenleistung der Threads, d.h. bei mehreren verfügbaren Kernen der CPU für jeden Thread, also das Auflaufen von Rohdatenobjekten in der ersten Queue verhindern würde. Eventuell könnte sogar auf diese Weise schon beim ersten Durchlauf der Leseroutine durch alle Dateien alle Berechnungen beendet sein. Infolgedessen würden sich auch die Wartezeiten des Schreib-Thread auf ein OutPutDatenKollektor-Objekt verkürzen. Ob weitere Rechnerthreads die Performance weiterhin erhöhen, müssen dann Zeitmessungen zeigen.

Eine weitere Verbesserung bezüglich der Performance könnte die Verbesserung der O-Klasse der Methode zum Glätten der X-Werte sein, welche in meiner Implementierung  $O(n^2)$  beträgt. So könnte eine andere Implementierung der Algorithmus-Schnittstelle z.B. für Performanceverbesserungen in dieser Hinsicht sorgen.

Weiterhin kann die Realisation der Datenvermittlung nach dem Einlesen wirklich genau im 50ms Takt erfolgen und keinen Schwankungen mehr unterliegen. Es bestünde wie in der Aufgabenbeschreibung schon angedeutet die Möglichkeit, einen weiteren Leser-Thread mit der Aufgabe zu betrauen, den Datenfluss der eingelesenen Informationen genau im 50ms – Takt „gleichzurichten“. Der erste Lesethread würde also eine gewisse Vorlaufzeit benötigen und die eingelesenen Objekte vorerst in einer „ArrayBlockingQueue“ von InputdatenKollektoren sammeln, bevor daraufhin der gleichrichtende zweite Lesethread in einer bezüglich der Abbruchbedingung identischen while()-Schleife alle 50ms pausiert, bevor die Weitergabe der Objekte an die Controller-Klasse bzw. an die Rechner-Threads erfolgt. Die Abbruchbedingung für beide Leserthread ist schließlich, dass alle neuartigen Daten in Dateien geschrieben wurden.

## Quellcode-Listing

```
package Main;

import Algorithmus.AKF_ALGO;
import Algorithmus.AKF_Loeser;
import Algorithmus.RechnerThread;
import Ausgabe.Schreiber;
import Ausgabe.SchreiberThread;
import Ausgabe.schreibInterface;
import Eingabe.*;
import Controller.Controler;
import Model.Exceptions.DatensatzInkonsistentException;
import Model.Exceptions.EingabeException;

import java.io.IOException;

/**
 * Das ist die Main-Klasse.
 * Sie beinhaltet die statische main() - Methode zum Starten des
 * gewünschten Softwaresystems.
 */
public class Main {

    /**
     * @param args Die Kommandozeilenparameter als String[] args.
     * Der erste String muss eine relative oder absolute
     * Pfadangabe zum Ordner der Testdateien darstellen. Ein zweiter String als
     * Angabe eines Zielordners ist optional.
     * @throws EingabeException Exception für
     * Fehlererscheinungen beim Einlesen zu Pfadangaben und Zugriffrechten.
     * @throws DatensatzInkonsistentException Exception zu fehlerhaft
     * strukturierten Datensätzen in eingelesenen Dateien.
     * @throws IOException Diese Exception zeigt in
     * unserem Fall auf, dass die Prozessausführung des Tools ABC außerhalb der
     * JVM aufgrund von falschen Parameterübergaben an das Tool oder einer
     * defekten Installation scheiterte.
     */
    public static void main(String[] args) throws EingabeException,
    DatensatzInkonsistentException, IOException, InterruptedException {

        if (args.length != 2 && args.length != 1) {
            System.err.println("Das Programm muss mit java -jar
<Programmpfad> <Eingabedatei> [<Ausgabedatei>] aufgerufen werden.");
            System.exit(0);
        }

        String zielOrdnerpfad = "";

        if (args.length == 1) {
            zielOrdnerpfad = args[0];
        } else {
            zielOrdnerpfad = args[1];
        }
    }
}
```

```

Controller ct = new Controller();
// datenSchnittstelleLeserThread dv = new DatenVermittler(ct);

LeseThread leserThread = new LeseThread(ct, args[0]);
leseInterface leser = new Leser();
leserThread.setLeser(leser);

RechnerThread rechnerThread = new RechnerThread(ct);
AKF_Loeser algo = new AKF_ALGO();
rechnerThread.setAlgo(algo);

    SchreiberThread schreiberThread = new SchreiberThread(ct,
zielOrdnerpfad);
    schreibInterface schreiber = new Schreiber();
    schreiberThread.setSchreiber(schreiber);

    final long timeStart = System.currentTimeMillis();
    leserThread.start();
    rechnerThread.start();
    schreiberThread.start();

    leserThread.join();
    rechnerThread.join();
    schreiberThread.join();

    final long timeEnd = System.currentTimeMillis();
    String time = "Verlaufszeit (mit Ausgabe): " + (timeEnd -
timeStart) + " Millisek.";
    System.out.println(" \n " + ".Berechnungen durchgefuehrt in " +
time);
    }
}

```

```

package Eingabe;

import Model.Exceptions.DatensatzInkonsistentException;
import Model.Exceptions.EingabeException;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Diese Klasse dient dem Einlesen der Daten aus der Quelldatei und dem
 * Initialisieren der Daten in passenden Datenstrukturen auf einer Instanz des
 * InputDatenKollektor.
 */
public class Leser implements leseInterface {
    private String quellPfad;

    /**
     * Gibt eine leere Leser-Instanz zurueck. Sie ist erst
     * funktionsfaehig, wenn der String quellPfad gesetzt wurde.
     */
    public Leser() {

    }

    /**
     * Die konkrete Methode zum Einlesen der Positionswerte des Spiegels
     * (X-Werte) und der Detektorwerte (Y-Werte).
     *
     * @throws EingabeException Diese Exception signalisiert
     * Fehler bei der Eingabe des Zielverzeichnis mit den Testdateien.
     * @throws DatensatzInkonsistentException Diese Exception signalisiert
     * inkonsistente Datensätze.
     */
    @Override
    public InputDatenKollektor liesEin(int id) throws EingabeException,
    DatensatzInkonsistentException {
        InputDatenKollektor inputDaten = new InputDatenKollektor();
        Scanner sc = null;
        final File eingabeDatei = new File(quellPfad);

        if (!eingabeDatei.exists() || !eingabeDatei.isFile()) {
            throw new EingabeException("Die Textdatei %s existiert nicht.",
            quellPfad);
        }

        try {
            sc = new Scanner(eingabeDatei);
        }
        // Keine Leserechte
        catch (FileNotFoundException e) {
            throw new EingabeException("Die Datei %s konnte nicht
            eingelesen werden: %s", quellPfad, e.getMessage());
        }

        int datenSatz_ct = 0;
        int kommentarZeilenCounter = 0;
    }

```

```

while (sc.hasNextLine()) {
    String line = sc.nextLine();
    if (istBeschreibungsZeile(line)) {
        ++kommentarZeilenCounter;
    }

    if (!istBeschreibungsZeile(line)) {
        ++datenSatz_ct;
        // Split am Delimiter Tabulator
        String[] line_arr = line.trim().split("\\t");

        try {
            if (line_arr.length != 2) {
                throw new DatensatzInkonsistentException("Es gibt
eine inkonsistente Daten-Zeile mit fehlenden Werten fuer X und Y in der
Datei \"" + id + ".txt\"!");
            }
        } catch (DatensatzInkonsistentException e) {
            System.out.println(e.getMessage());
            return null;
        }

        try {
            int x = Integer.parseInt(line_arr[1]);
            inputDaten.getxWerte().add(x);
        } catch (NumberFormatException e) {
            System.out.println("X_Wert liegen in inkorrektem
Datentyp vor in der Datei \"" + id + ".txt\" " + e.getMessage());
            return null;
        }

        try {
            int y = Integer.parseInt(line_arr[0]);
            if (y < 0) {
                throw new DatensatzInkonsistentException("Es
existieren negative Y-Werte kleiner 0 in der Datei \"" + id + ".txt\", was
nicht dem Sachverhalt entsprechen kann.");
            } else {
                inputDaten.getyWerte().add(y);
            }
        } catch (NumberFormatException e) {
            System.out.println("Y_Wert liegen in inkorrektem
Datentyp vor in der Datei \"" + id + ".txt\" " + e.getMessage());
            return null;
        } catch (DatensatzInkonsistentException e) {
            System.err.println(e.getMessage());
            sc.close();
            return null;
        }
    }
}

try {
    if (datenSatz_ct <= 0) {
        throw new DatensatzInkonsistentException("Fehler: Die Datei
\"" + id + ".txt\" enthaelt keine keinerlei Datensaeetze!"); // eigene
Exception
    }
} catch (DatensatzInkonsistentException e) {
    System.err.println(e.getMessage());
    sc.close();
    return null;
}

```



```

    }
    if (kommentarZeilenCounter != 1) {
        System.out.println("Die Datei Die Datei \"" + id + ".txt\"
besitzt keine Kommentarzeile, wird aber verarbeitet.");
    }
    sc.close();

    inputDaten.setDatenID(id);

    return inputDaten;
}

/**
 * @param _ordnerPfad Setzt den Pfad zum Zielordner mit den Dateien in
dem das Interface implementierenden Leser-Objekt.
 */
@Override
public void setOrdnerPfad(String _ordnerPfad) {
    this.quellPfad = _ordnerPfad;
}

/**
 * @param line einzulesende Zeile. Methode prueft, obt diese Zeile eine
Kommentarzeile ist.
 * @return
 */
private boolean istBeschreibungsZeile(String line) {
    return line.startsWith("#");
}
}

```

```

package Eingabe;

import Model.Exceptions.DatensatzInkonsistentException;
import Model.Exceptions.EingabeException;

/**
 * Das Interface fuer die Implementierung des Leseverhaltens der Leser-
 * instanzen.
 */
public interface leseInterface {
    /**
     * @param id Die Id-des eingelesenen Datensatzes wird "von aussen"
     * mitgeteilt, da sie im Dateinamen steht.
     * @return
     * @throws EingabeException Wird bei untauglichen
     * Benutzereingaben bezueglich des Ordners der Eingabedateien geworfen.
     * @throws DatensatzInkonsistentException Wird bei inkonsistenten
     * Datensatzen geworfen.
     */
    InputDatenKollektor liesEin(int id) throws EingabeException,
    DatensatzInkonsistentException;

    /**
     * @param _ordnerPfad Setzt den Pfad zum Zielordner mt den Dateien in
     * dem das Interface implementierenden Leser-Objekt.
     */
    void setOrdnerPfad(String _ordnerPfad);
}

```

```

package Eingabe;

import Controller.Controler;
import Model.DatenObjekt;
import Model.Exceptions.DatensatzInkonsistentException;
import Model.Exceptions.EingabeException;

import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import java.util.HashSet;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Set;

/**
 * Diese Lesethread-Klasse leistet in seiner run-Methode das fortlaufende
 * Einlesen und die Uebertragung der Inhalte 10 Dateien als Endlosschleife.
 */
public class LeseThread extends Thread {
    private Controler ct;
    private String ordnerpfad;
    private final Set dateiNamen;
    private InputDatenKollektor inPutPuffer; // für das regelmaeßige zur
    Verfuegung stellen im 0.05 Sekunden-Takt
    private leseInterface leser;

    public LeseThread(Controler _ct, String _ordnerpfad) throws IOException
    {
        this.ct = _ct;
        this.ordnerpfad = _ordnerpfad;
        this.dateiNamen = listeDateien(ordnerpfad);
    }

    /**
     * Diese Methode wird im Thread zum kontinuierlichen Einlesen
     * Datensätzen ausgefuehrt.
     * Abbruchbedingung für den Thread ist, falls in der Controler Klasse
     * das flag "alleArbeitGemacht" auf true steht.
     */
    @Override
    public void run() {

        Iterator<String> pathIterator = dateiNamen.iterator();
        while (!ct.alleDateienGeschrieben.get()) {
            String naechsterPfad = "";
            if (!pathIterator.hasNext()) {
                pathIterator = dateiNamen.iterator();
            }
            try {
                naechsterPfad = pathIterator.next();
            } catch (NoSuchElementException e) {
                System.out.println("Der angegebene Ordner enthaelt
keinerlei Dateien. Programmende.");
                System.exit(0);
            }
        }
    }
}

```

```

        String[] idArr = naechsterPfad.trim().split("\\.");
        leser.setOrdnerPfad(ordnerpfad + "/" + naechsterPfad);
        try {
            this.inPutPuffer =
leser.liesEin(Integer.parseInt(idArr[0])); // Hier findet das
Überschreiben der vorherigen Daten im inPutPuffer-Attribut statt.
            if (this.inPutPuffer == null) {
                dateiNamen.remove(naechsterPfad);
                pathIterator = dateiNamen.iterator();
                if (dateiNamen.size() == 0) {
                    throw new DatensatzInkonsistentException("Alle
Dateien sind inkonsistent. Beende Simulation!");
                }
            } else {
                this.ct.addDatenObjekt(new DatenObjekt(inPutPuffer));
            }
        } catch (DatensatzInkonsistentException | EingabeException e) {
            System.out.println(e.getMessage());
            System.exit(0);
        }
    }

    /**
     * Diese Methode nutzt einen Directory-Stream, um ueber alle Eintraege
     im angegebenen Ordner zu iterieren um alle Path-Objekte als String
     abspeichern zu koennen.
     *
     * @param ordner Der Pfad zum Zielordner.
     * @return Ein Set von Strings mit allen im Ordner befindlichen
     Dateinamen.
     * @throws IOException
     */
    private Set<String> listeDateien(String ordner) throws IOException {
        Set<String> dateiListe = new HashSet<>();
        try (DirectoryStream<Path> stream =
Files.newDirectoryStream(Paths.get(ordner))) {
            for (Path pfad : stream) {
                if (!Files.isDirectory(pfad) &&
!pfad.toString().contains("out")) {
                    dateiListe.add(pfad.getFileName()
.toString());
                }
            }
        } catch (Exception e) {
            System.out.println("Fehler:" + e.getMessage() + " ist kein
gueltiger Ornder mit Inputdateien! Beende Simulation");
            System.exit(0);
        }
        return dateiListe;
    }

    /**
     * Ein Setter fuer eine benutzerspezifische Leser Instanz.
     *
     * @param _leser Eine Instanz eines benutzerspezifischen Lesers, welche
     das leseInterface implementieren muss.
     */

```

```
public void setLeser(leseInterface _leser) {  
    this.leser = _leser;  
}  
}
```

```

package Eingabe;

import java.util.ArrayList;

/**
 * Eine reine Datenhaltungsklasse zur Datenspeicherung der aufbereiteten
 * Daten aus der Eingabedatei in entsprechenden Datenstrukturen.
 * Diese Klasse ist die Grundlage zur Weiterverarbeitung der Daten.
 */
public class InputDatenKollektor {

    private ArrayList<Integer> xWerte;
    private ArrayList<Integer> yWerte;
    private int datenID = -1;

    /**
     * Die Member werden zuerst "leer" initialisiert und dann, falls das
     * Einlesen erfolgreich verlief, durch die Leser-Instanz von außen "gesetzt".
     */
    public InputDatenKollektor() {
        xWerte = new ArrayList<>();
        yWerte = new ArrayList<>();
    }

    public InputDatenKollektor(InputDatenKollektor _inputDatenKollektor) {
        this.xWerte = _inputDatenKollektor.getXWerte();

        this.yWerte = _inputDatenKollektor.yWerte;
        this.datenID = _inputDatenKollektor.datenID;
    }

    // es folgen alle noetigen Setter und Getter fuer die Liste der X,Y-
    // Werte und die datenID.
    public ArrayList<Integer> getXWerte() {
        return xWerte;
    }

    public void setxWerte(ArrayList<Integer> xWerte) {
        this.xWerte = xWerte;
    }

    public ArrayList<Integer> getYWerte() {
        return yWerte;
    }

    public void setyWerte(ArrayList<Integer> yWerte) {
        this.yWerte = yWerte;
    }

    public int getDatenID() {
        return datenID;
    }

    public void setDatenID(int datenID) {
        this.datenID = datenID;
    }
}

```

```

package Algorithmus;

import Eingabe.InputDatenKollektor;
import Model.FHWM;

import java.util.ArrayList;

/**
 * Die Klasse mit der Implementierung der vier Algorithmen zur X-Wert-
 * Glaettung, Normierung der Y-Werte, Bestimmung approximierter Werte zur
 * einhuellenden Funktion und die Pulsbreitenbestimmung.
 */
public class AKF_ALGO implements AKF_Loeser {

    private InputDatenKollektor inPut;
    private int N;
    private ArrayList<Double> xPiko;
    private ArrayList<Double> xGlatt;
    private ArrayList<Double> YNorm;
    private double[] huellende;
    private FHWM fhwm;
    private int yMaxAbs;
    private int idxYMax;
    private int aktuelleDatenID;

    /**
     * Dieser Konstruktor erzeugt nur eine leere Instanz ohne die nötigen
     * Inputdaten. Er ist noetig bei der Instanziierung der Loeserthread-Klasse,
     * bevor deren run() Methode aufgerufen bzw. der Lese-Thread gestartet
     * wird. Ind diesem Vorgang wird die Algo-Klasseninstanz per Setter
     * funktionsfaehig.
     */
    public AKF_ALGO() {
        xPiko = new ArrayList<Double>();
        xGlatt = new ArrayList<Double>();
        YNorm = new ArrayList<Double>();
        fhwm = new FHWM();
    }

    /**
     * Bestimmt wie in der Aufgabe vorgegeben die geglaetteten X-Werte aus
     * der Eingabedatei.
     * Hat eine Laufzeitkomplexitaet von  $O(n^2)$  aufgrund zweier
     * verschachtelter Schleifen.
     */
    @Override
    public void bestimmeXk() {
        this.N = inPut.getxWerte().size(); // auch gleich der size() von
        inPut.yWerte

        for (int i = 0; i < N; i++) {
            xPiko.add((((inPut.getxWerte().get(i) * ((266.3 / (Math.pow(2,
18) - 1)))) - 132.3)));
        }

        int n = bestimme_n();
        int tau = (n - 1) / 2;

        for (int k = 0; k < N; k++) {

```

```

        double xsum = 0;
        boolean linkerRand = false;
        boolean rechterRand = false;
        int ct = 0; // Counter fuer die Randfaelle der Mittelung, das
        // das Mittelungsfenster sich in den Randfaellen aendert.
        for (int i = 0; i < n; i++) {
            if (k < tau) {
                linkerRand = true;
                xsum += xPiko.get(i);
                ++ct;
            } else if (k > N - 1 - tau) {
                rechterRand = true;
                xsum += xPiko.get(N - 1 - i);
                ++ct;
            } else {
                xsum += xPiko.get(k - tau + i);
            }
        }
        if (linkerRand || rechterRand) {
            xGlatt.add((xsum / ct));
        } else {
            xGlatt.add((xsum / n));
        }
    }

    /**
     * Normiert alle erhaltenen absoluten Y-Werte (Detektorwerte);
     * Laufzeitkomplexitaet O(n)
     */
    @Override
    public void normiereY() {
        int y_Max = 0;
        for (int i = 0; i < N; i++) {
            if (inPut.getyWerte().get(i) > y_Max) {
                y_Max = inPut.getyWerte().get(i);
                idxYMax = i;
            }
        }
        this.yMaxAbs = y_Max;
        for (int z : inPut.getyWerte()) {
            YNorm.add((z / (double) y_Max));
        }
    }

    /**
     * Bestimmt die Eintraege zur huellendenKurve. Hat eine Laufzeit von
     * O(n).
     */
    @Override
    public void bestimmeHuellende() {
        double lokalesMaxYlinks = 0;
        for (int i = 0; i < N; i++) {
            if (this.YNorm.get(i) > lokalesMaxYlinks) {
                lokalesMaxYlinks = this.YNorm.get(i);
            }
            this.huellende[i] = lokalesMaxYlinks;
            if (i == idxYMax) {
                break;
            }
        }
    }

```



```

    }

    double lokalesMaxYrechts = 0;

    for (int i = N - 1; i >= 0; i--) {
        if (this.YNorm.get(i) > lokalesMaxYrechts) {
            lokalesMaxYrechts = this.YNorm.get(i);
        }

        this.huellende[i] = lokalesMaxYrechts;
        if (i == idxYMax) {
            break;
        }
    }
}

/**
 * @return Als Rueckgabe wird hier ein FWHM-Objekt mit den passenden
Werten erstellt.
 * Als erstes wird die Grundlinie-Hoehe in der Menge der normierten Y-
Werte bestimmt. Anschliessend die zur Haelfte des Abstandes (1-Grundlinie)
zugehörigen Indize idxL
 * und idxR und die zugehoerige Distanz bzw. Pulsbreite in den
geglatteten X-Werten.
 * Laufzeit O(n).
 */
@Override
public void bestimmeFWHM() {

    int indexErstesProzent = (int) (N * 0.01);

    double sumErstesProzent = 0;
    for (int i = 0; i < indexErstesProzent; i++) {
        sumErstesProzent += huellende[i];
    }

    double grundlinie = sumErstesProzent / indexErstesProzent;

    double halb = ((1 + grundlinie) * 0.5);

    int idxL = 0;
    int idxR = 0;

    for (int i = 0; i < N; i++) {
        if (huellende[i] > halb) {
            idxL = i;
            break;
        }
    }

    for (int i = N - 1; i > 0; i--) {
        if (huellende[i] > halb) {
            idxR = i;
            break;
        }
    }

    double b = (Math.abs(xGlatt.get(idxR) - xGlatt.get(idxL)));

```

```

        this.fhwm.setFHWm(b);
        this.fhwm.setL(idxL);
        this.fhwm.setR(idxR);
    }

    /**
     * @return Bestimmt mit der ungeraden positiven Ganzzahl int n die
     Groesse des Mittelungsfenster.
     */
    @Override
    public int bestimme_n() {
        int ret = 0;
        if (((int) Math.floor(0.002 * this.N)) % 2 == 0) {
            ret = (int) Math.floor(0.002 * this.N) - 1;
        } else
            ret = (int) Math.floor(0.002 * this.N);

        return ret;
    }

    /**
     * @return Gibt nach dem Aufruf aller 4 Algorithmen ein
     OutputDatenKollektor-Objekt mit allen zur Ausgabe noetigen Daten zurueck.
     */
    @Override
    public OutputDatenKollektor holeLoesungsmengen() {
        FHWm fhwm_clone = new FHWm(this.fhwm);
        ArrayList<Double> YNorm_clone = (ArrayList<Double>)
this.YNorm.clone();
        double[] huellende_clone = this.huellende.clone();
        ArrayList<Double> xGlatt_clone = (ArrayList<Double>)
this.xGlatt.clone();

        return new OutputDatenKollektor(fhwm_clone, xGlatt_clone,
YNorm_clone, huellende_clone, this.aktuelleDatenID);
    }

    /**
     * In dieser wichtigen Methode werden die Rohdaten pro neuem Objekt aus
     der Rohdaten-Queue in der Instanz dieser Algorithmusklassse gesetzt und die
     Insatnz
     * fuer einen neuen Bearbeitungsvorgang dieser Daten vorbereitet.
     *
     * @param _inPut Eine Instanz von InputDatenKollektor mit Rohdaten aus
     einer Datei.
     */
    @Override
    public void setInput(InputDatenKollektor _inPut) {
        this.inPut = new InputDatenKollektor(_inPut);
        this.huellende = new double[this.inPut.getyWerte().size()];
        xPiko = new ArrayList<Double>();
        xGlatt = new ArrayList<Double>();
        YNorm = new ArrayList<Double>();
        aktuelleDatenID = _inPut.getDatenID();
        fhwm = new FHWm();
    }
}

```

```

package Algorithmus;

import Eingabe.InputDatenKollektor;

/**
 * Das Interface zur Implementierung der mathematischen Algorithmen:
 * Glaetten der X-Werte, Normierung der Y-Werte, Bestimmungen der
 * Huellendenkurve und der Pulsbreite.
 */
public interface AKF_Loeser {

    /**
     * Rechnet die X-Eingabewerte zuerst in Pikosekunden um und "glaettet"
     die Werte anschließend
     */
    void bestimmeXk();

    /**
     * Normiert die gegebenen Y-Werte, so dass diese im Intervall (0,1]
     liegen.
     */
    void normiereY();

    /**
     * Traegt für die Simulation der oberen Einhüllenden entsprechende
     "zuletzt" maximale Y-Wert aus der Liste der normierten Y-Werte auf.
     */
    void bestimmeHuellende();

    /**
     * @return ein FWHM-Objekt, welches die relevanten Objekte zur
     Pulsbreitenbestimmung kapselt.
     */
    void bestimmeFWHM();

    /**
     * @return die ungerade Ganzzahl n, welche zur Berechnung der Glaettung
     erforderlich ist.
     */
    int bestimme_n();

    /**
     * @return OutPutDatenKollektor. Gibt nach erfolgreicher Berechnung
     aller 4 Algorithmen ein Objekt mit den gekapselten Loesunungsmengen zurueck.:
     */
    OutputDatenKollektor holeLoesungsMengen();

    /**
     * Ein Setter für die Rohdatenverarbeitung in der Algorithmusklasse.
     *
     * @param _inPut Eine Instanz von InputDatenKollektor mit Rohdaten aus
     einer Datei.
     */
    void setInput(InputDatenKollektor _inPut);
}

```

```

package Algorithmus;

import Controller.Controler;
import Eingabe.InputDatenKollektor;

import java.util.concurrent.TimeUnit;

/**
 * Die Kernklasse meines Problemloesers. Sie ist das "Zentrum": Stellt eine
 * Instanz des Output-DatenKollektoren bereit, auf denen ein per Strategy-
 * Pattern gewählte Implementierung
 * der 4 Loesungsalgorithmen arbeitet.
 * Sie besitzt Instanzen der Schnittstellenimplementierungen
 * "schreibInterface" und "leseInterface" um flexibel auf Aenderungen in den
 * Ein- und Ausgabeformaten der Eingabe und Ausgabedaten reagieren zu koennen.
 */

public class RechnerThread extends Thread {
    private AKF_Loeser algo;
    private OutputDatenKollektor outPutKollekt;
    private final Controler ct;

    /**
     * Erstellt eine noch nicht funktionsfaehige Loeserthread-Instanz. Erst
     * nach dem Setzen mit einer Algorithmus-Instanz kann der Thread gestartet
     * werden.
     */
    public RechnerThread(Controler _ct) {
        this.ct = _ct;
    }

    /**
     * Die run()-Methode des zu startenden Loeserthreads. Sie arbeitet
     * solange mit Rohdatenobjekte aus der Inputdaten-Queue im Controler ab, bis
     * mit Sicherheit keine neuartigen Rohdaten mehr erwartet werden koennen.
     */
    @Override
    public void run() {

        while (true) {
            if (this.ct.alleDatenGeliefert.get()) {
                System.out.println("Alle daten aufbereitet");
                this.ct.alleDatenAufbereitet.set(true);
                return;
            }
            InputDatenKollektor neuerInput = null;
            try {
                neuerInput =
this.ct.getDetektordatenQueue().take().getInPut();
            } catch (InterruptedException e) {
                //e.printStackTrace();
            }
            this.algo.setInput(neuerInput);
            final long timeStart = System.currentTimeMillis();
            algo.bestimmeXk();
            algo.normiereY();
            algo.bestimmeHuellende();

```

```

        algo.bestimmeFWHM();
        this.outPutKollekt = algo.holeLoesungsMengen();
        try {
            this.ct.getOutputDatenQueue().put(outPutKollekt);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        final long timeEnd = System.currentTimeMillis();
        String s = "Verlaufszeit der Berechnung: Fuer den Datensatz
Nr." + neuerInput.getDatenID() + " = " + (timeEnd - timeStart) + "
Millisek.";
        System.out.println(" \n " + s);

    }

}

/**
 * Ein Setter für eine bestimmte Implementierung eines gewünschten
Loesungsalgorithmus.
 */
 * @param _algo Eine Instanz mit einer Implementierung der
Algorithmusschnittstelle "LoesungsAlgo" zur Problemloesung.
 */
public void setAlgo(AKF_Loeser _algo) {
    this.algo = _algo;
}

}

```

```

package Algorithmus;

import Model.FHWM;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Locale;

/**
 * Eine reine Klasse zur Datenhaltung.
 * Sammelt alle relevanten Loesungsdaten in den angegebenen Membern, falls
 * der LoeserThread einen Datensatz erfolgreich verarbeitet.
 */
public class OutputDatenKollektor {
    private FHWM fhwm;
    private String fhwmDatenZeile;
    private String spaltenHeader;
    private ArrayList<Double> normierteIntensitaetYN;
    private ArrayList<Double> xGlatt;
    private double[] huellendenWerte;
    private int datenID;

    /**
     * Erzeugt einen im Sinne der Ausgabe vollstendigen Outputdaten-
     * Kollektor.
     */
    public OutputDatenKollektor(FHWM _fhwm, ArrayList<Double> _xGlatt,
        ArrayList<Double> _normierteIntensitaet, double[] _huellendenWerte, int
        datenID) {
        this.fhwm = _fhwm;
        this.fhwmDatenZeile = "# " + "FWHM = " + String.format(Locale.US,
            "%.16e", this.fhwm.getFhwm()) + ", " + this.fhwm.getL() + ", " +
            this.fhwm.getR();
        this.spaltenHeader = "# " + "pos " + "int\t" + "env\n";
        this.normierteIntensitaetYN = _normierteIntensitaet;
        this.huellendenWerte = _huellendenWerte;
        this.xGlatt = _xGlatt;
        this.datenID = datenID;
    }

    /**
     * @return Gibt die FHWM Werte als String-Datenzeile zurueck.
     */
    public String getFhwmDatenZeile() {
        return fhwmDatenZeile;
    }

    /**
     * @return Gibt die Ueberschrift aller Ausgabespalten als String
     * zurueck.
     */
    public String getSpaltenHeader() {
        return spaltenHeader;
    }

    /**
     * @param spaltenHeader Setzt die Spaltenueberschriften durch einen
     * uebergegebenen String.
     */
}

```

```

    public void setSpaltenHeader(String spaltenHeader) {
        this.spaltenHeader = spaltenHeader;
    }

    /**
     * @return Gibt die Liste mit den normierten Intensitaetswerten
     * zurueck.
     */
    public ArrayList<Double> getNormierteIntensitaetYN() {
        return normierteIntensitaetYN;
    }

    /**
     * @return Gibt das double-Array mit den Werten der approximierten
     * Werten der Einhuellendenfunktion der Y-Werte zurueck.
     */
    public double[] getHuellendenWerte() {
        return huellendenWerte;
    }

    /**
     * @return Gibt die Liste der geglaetteten X-Werte zurueck.
     */
    public ArrayList<Double> getXGlatt() {
        return xGlatt;
    }

    /**
     * @return Gibt die jeweilige Datensatz-ID als Integer zurueck.
     */
    public int getDataenID() {
        return datenID;
    }
}

```

```

package Controller;

import Algorithmus.OutputDatenKollektor;
import Model.DatenObjekt;

import java.util.HashSet;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * Diese Klasse dient als Schnittstelle zur Datenhaltung und Kontrolle
 * neuer Datensätze vom Leser-Thread, bevor sie vom Loeserthread
 * weiterverarbeitet werden.
 * Auch der Schreiber-Thread erhält von hier aus seine nötigen
 * Outputdaten. Beides wird über Queue-Implementierungen realisiert.
 * Sie enthält weiterhin flags, die signalisieren, ob DetektorDaten nicht
 * vorliegen, alle Datenobjekte verarbeitet wurden oder alle Dateien
 * geschrieben wurden.
 */
public class Controller {

    private final ArrayBlockingQueue<DatenObjekt>
queue_gesammelteDetektordaten;
    private final ArrayBlockingQueue<OutputDatenKollektor> queue_output;
    private final HashSet<Integer> idSet;
    public AtomicBoolean alleDateienGeschrieben = new AtomicBoolean(false);
    public AtomicBoolean alleDatenAufbereitet = new AtomicBoolean(false);
    public AtomicBoolean alleDatenGeliefert = new AtomicBoolean(false);

    /**
     * Initialisiert die leeren Queues.
     */
    public Controller() {
        this.queue_gesammelteDetektordaten = new ArrayBlockingQueue<>(4);
        this.queue_output = new ArrayBlockingQueue<>(10);
        this.idSet = new HashSet<>();
    }

    /**
     * @param dto Ein DatenObjekt dto mit allen zur Verarbeitung nötigen
     * X- und Y-Werten, welches dem Loeserthread in einer queue zur Verfügung
     * gestellt wird.
     * Es wird nur dann in die queue eingefügt, falls es ein
     * wirklich ein neuartiger Datensatz vom LeseThread "geliefert" wird.
     * Dies wird über ein Hashset kontrolliert, welches die IDs
     * der Datensätze sammelt, sodass neuartige IDs bzw. auch einen neuen
     * Datensatz für die Queue anzeigen.
     * @return
     */
    public boolean addDatenObjekt(DatenObjekt dto) {
        if (!idSet.contains(dto.datenID)) {
            try {
                while (!this.queue_gesammelteDetektordaten.offer(dto, 20,
TimeUnit.MILLISECONDS)) {
                }
                idSet.add(dto.datenID);
                return true;
            }
        }
    }
}

```



```

        } catch (InterruptedException interruptedException) {
            System.out.println(interruptedException.getMessage());
        }
    } else {

        this.alleDatenGeliefert.set(true);
    }
    return false;
}

/**
 * @return Legt eine Referenz auf die gesammelten Detektordaten bzw.
 * Inputdaten zum zugriff waehrend der Verarbeitung frei.
 */
    public synchronized ArrayBlockingQueue<DatenObjekt>
getDetektordatenQueue() {
        return this.queue_gesammelteDetektordaten;
    }

/**
 * @return Legt eine Referenz auf die gesammelten Outputdaten bzw.
 * Inputdaten zum Zugriff waehrend des Schreibvorgangs frei.
 */
    public synchronized ArrayBlockingQueue<OutputDatenKollektor>
getOutputDatenQueue() {
        return this.queue_output;
    }
}

```

```

package Controller;

import Algorithmus.OutputDatenKollektor;
import Model.DatenObjekt;

import java.util.HashSet;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * Diese Klasse dient als Schnittstelle zur Datenhaltung und Kontrolle
 * neuer Datensätze vom Leser-Thread, bevor sie vom Loeserthread
 * weiterverarbeitet werden.
 * Auch der Schreiber-Thread erhält von hier aus seine nötigen
 * Outputdaten. Beides wird über Queue-Implementierungen realisiert.
 * Sie enthält weiterhin flags, die signalisieren, ob DetektorDaten nicht
 * vorliegen, alle Datenobjekte verarbeitet wurden oder alle Dateien
 * geschrieben wurden.
 */
public class Controller {

    private final ArrayBlockingQueue<DatenObjekt>
queue_gesammelteDetektordaten;
    private final ArrayBlockingQueue<OutputDatenKollektor> queue_output;
    private final HashSet<Integer> idSet;
    public AtomicBoolean alleDateienGeschrieben = new AtomicBoolean(false);
    public AtomicBoolean alleDatenAufbereitet = new AtomicBoolean(false);
    public AtomicBoolean alleDatenGeliefert = new AtomicBoolean(false);

    /**
     * Initialisiert die leeren Queues.
     */
    public Controller() {
        this.queue_gesammelteDetektordaten = new ArrayBlockingQueue<>(4);
        this.queue_output = new ArrayBlockingQueue<>(10);
        this.idSet = new HashSet<>();
    }

    /**
     * @param dto Ein DatenObjekt dto mit allen zur Verarbeitung nötigen
     * X- und Y-Werten, welches dem Loeserthread in einer queue zur Verfügung
     * gestellt wird.
     * Es wird nur dann in die queue eingefügt, falls es ein
     * wirklich ein neuartiger Datensatz vom LeseThread "geliefert" wird.
     * Dies wird über ein Hashset kontrolliert, welches die IDs
     * der Datensätze sammelt, sodass neuartige IDs bzw. auch einen neuen
     * Datensatz für die Queue anzeigen.
     * @return
     */
    public boolean addDatenObjekt(DatenObjekt dto) {
        if (!idSet.contains(dto.datenID)) {
            try {
                while (!this.queue_gesammelteDetektordaten.offer(dto, 20,
TimeUnit.MILLISECONDS)) {
                }
                idSet.add(dto.datenID);
                return true;
            }
        }
    }
}

```

```

        } catch (InterruptedException interruptedException) {
            System.out.println(interruptedException.getMessage());
        }
    } else {

        this.alleDatenGeliefert.set(true);
    }
    return false;
}

/**
 * @return Legt eine Referenz auf die gesammelten Detektordaten bzw.
 * Inputdaten zum zugriff waehrend der Verarbeitung frei.
 */
    public synchronized ArrayBlockingQueue<DatenObjekt>
getDetektordatenQueue() {
        return this.queue_gesammelteDetektordaten;
    }

/**
 * @return Legt eine Referenz auf die gesammelten Outputdaten bzw.
 * Inputdaten zum Zugriff waehrend des Schreibvorgangs frei.
 */
    public synchronized ArrayBlockingQueue<OutputDatenKollektor>
getOutputDatenQueue() {
        return this.queue_output;
    }
}

```

```

package Ausgabe;

import Algorithmus.OutputDatenKollektor;
import Model.Exceptions.Ausgabe_Exception;

import java.io.IOException;

/**
 * Das Interface zur Implementierung des Schreibverhaltens.
 */
public interface schreibInterface {
    /**
     * Dies ist die Methodensignatur zur Methode, welche in Dateien
     schreibt. Implementiert in Schreiber-Instanzen.
     *
     * @throws Ausgabe_Exception kann geworfen werden, falls Fehler beim
     Schreiben auftreten.
     */
    void schreibeInDatei() throws Ausgabe_Exception, IOException;

    /**
     * Dies ist die Methodensignatur zur Methode, welche alle noetigen
     Daten fuer den Schreibvorgang festlegt. implementiert in Schreiber-
     Instanzen.
     *
     * @param _zielPfad Der zielPfad zum Ordner, in dem die Ausgabedateien
     zu schreiben sind.
     * @param _out Die Datencontainerklasse, welche die die
     auszugebenden Inhalte enthaelt.
     */
    void setSchreibDaten(String _zielPfad, OutputDatenKollektor _out);
}

```

```

package Ausgabe;

import Controller.Controler;
import Model.Exceptions.Ausgabe_Exception;

import java.io.IOException;

/**
 * Die Klasse, welche die run()-Methode des Schreiber-thread implementiert.
 */
public class SchreiberThread extends Thread {
    private Controler ct;
    private String _zielordnerPfad;
    private schreibInterface schreiber;

    /**
     * @param _ct Eine Referenz auf die Instanz der zentrale
     Datenklasse Controler, innerhalb derer Input- und output-Datensaetze
     "verteilt" werden.
     * @param zielOrdnerPfad Der Pfad zum Zielordner, welcher entweder mit
     dem Ordner der Eingabedateien identisch oder optional angegeben werden
     kann.
     */
    public SchreiberThread(Controler _ct, String zielOrdnerPfad) {
        this._zielordnerPfad = zielOrdnerPfad;
        this.ct = _ct;
    }

    /**
     * Die Implementierung der run() - Methode des fuer das Schreiben in
     Dateien zustandigen Threads. Hier werden fuer jeden
     * vorhandenen Output-Datensatz Schreiber-Instanzen erzeugt, welche auf
     die Outputdaten per Referenz zugreifen.
     */
    @Override
    public void run() {
        while (!ct.alleDateienGeschrieben.get()) {
            if (ct.alleDatenAufbereitet.get() &&
ct.alleDatenGeliefert.get()) {
                ct.alleDateienGeschrieben.set(true);
            }
            try {
                schreiber.setSchreibDaten(_zielordnerPfad,
ct.getOutputDatenQueue().take()); // Pfad durchreichen;
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                schreiber.schreibeInDatei();
            } catch (Ausgabe_Exception | IOException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * Ein Setter für einen benutzerspezifischen Schreiber.
     *
     * @param _schreiber Eine Objektinstanz einer benutzerspezifischen
     Schreiber Instanz, welche das schreibInterface implementieren muss.
     */
}

```

```
    */  
    public void setSchreiber(schreibInterface _schreiber) {  
        this.schreiber = _schreiber;  
    }  
}
```

```

package Model;

import Eingabe.InputDatenKollektor;

import java.util.ArrayList;

/**
 * Ein DatenObjekt zur Kapselung der InputDaten aus einer Datei, um sie in
 * dieser Form im Controller zu in einer queue zu speichern.
 */
public class DatenObjekt {
    private InputDatenKollektor inPut;
    public int datenID;

    /**
     * Hier werden alle relevanten Member eines InputDatenKollektors als
     * echte Kopien gespeichert.
     *
     * @param _inPut eine Instanz vom InputDatenKollektor, welche per Copy-
     * Konstruktor im DatenObjekt als echte Kopie gespeichert wird.
     */
    public DatenObjekt(InputDatenKollektor _inPut) {
        inPut = _inPut;
        this.datenID = _inPut.getDatenID();
    }

    /**
     * @return Legt eine Referenz auf den InputDatenKollektor frei, um die
     * Daten erneut abfragen zu koennen.
     */
    public InputDatenKollektor getInPut() {
        return this.inPut;
    }
}

```

```

package Model;

/**
 * Eine Klasse zur Kapselung aller fuer die Pulsbreite relevanter Daten.
 */
public class FHWM {
    private double fhwm;
    private int L;
    private int R;

    public FHWM() {
        this.fhwm = 0;
        this.L = 0;
        this.R = 0;
    }

    /**
     * @param _fhwm Copy-Konstruktor fuer echte Kopien.
     */
    public FHWM(FHWM _fhwm) {
        this.setFHWM(_fhwm.getFhwm());
        this.setR(_fhwm.getR());
        this.setL(_fhwm.getL());
    }

    // alle noetigen Getter und Setter fuer die Attribute.

    public double getFhwm() {
        return fhwm;
    }

    public void setFHWM(double _fhwm) {
        this.fhwm = _fhwm;
    }

    public int getL() {
        return L;
    }

    public void setL(int l) {
        L = l;
    }

    public int getR() {
        return R;
    }

    public void setR(int r) {
        R = r;
    }
}

```



```

package Model.Exceptions;

/**
 * Eine Exception-Klasse fue fehlerhafte Ausgabefaelle.
 */
public class Ausgabe_Exception extends Exception {

    /**
     * @param s
     */
    public Ausgabe_Exception(String s) {
        super(s);
        System.err.println(s);
    }

    /**
     * @param message
     * @param replace
     */
    public Ausgabe_Exception(String message, String... replace) {
        super(String.format(message, replace));
        System.err.println(String.format(message, replace));
    }
}

```

```

package Model.Exceptions;

/**
 * Eine Exceptionklasse fuer inkonsistente Datensatze.
 */
public class DatensatzInkonsistentException extends Exception {

    public DatensatzInkonsistentException(String s) {
        super(s);
        System.err.println(s);
    }

}

```

```

package Model.Exceptions;

/**
 * Eine Exceptionklasse fuer fehlerhafte Benutzereingaben.
 */
public class EingabeException extends Exception {

    public EingabeException(String s) {
        super(s);
        System.err.println(s);
    }

    public EingabeException(String message, String... replace) {
        super(String.format(message, replace));
        System.err.println(String.format(message, replace));
    }

}

```