Aufgabe 4: Fahrradwerkstatt

Team-ID: ?????

Team-Name: ????

Bearbeiter/-innen dieser Aufgabe:
Tony Borchert

November 21, 2022

Contents

1 Teilaufgabe 2

Es werden nicht alle Kunden zufrieden sein, da der Auftrag eines Kunden A, der vor anderen Kunden einen Auftrag vergibt, nicht um bedingt als erstes bearbeitet wird. Dies passiert, wenn der Auftrag von einem Kunden nicht als einziger eingegangener Auftrag zur Verfügung steht und wenn die anderen Aufträge kürzer sind, aber nach dem des Kunden eingegangen sind. Im schlimmsten Fall, ist der Kunde mit seinem Auftrag sogar als letztes dran.

2 Teilaufgabe 3

Wenn man in einem echten Geschäft bestimmte Kunden einfach vorlassen würden, würden sich auch die Kunden die nicht vorgezogen werden verständlich beschweren. Denn egal wann oder unter welchen Umständen man einen Kunden A vor einem anderen Kunden B lässt, es dauert für B immer länger.

Nur in Ausnahmen lässt man einen Kunden nach vorne. So könnte auch Marc entscheiden, dass ein ganz kurzer Auftrag vor einem ganz langen Auftrag geschoben wird, wenn es die Wartezeit des langen Auftrages kaum vergrößert und der lange Auftrag noch nicht viel warten musste. Wo aber die Grenze genau ist, ist arbiträr, mathematisch nicht belegt und sollte deshalb vom Verkäufer stammen, möglicherweise auch wie er seine Kunden einschätzt. Deswegen kann dies nicht algorithmisch dargestellt werden.

Eine weitere Ausnahme könnte liegen, wenn es sich um einen Kunden handelt, der von Marc besonders geschätzt wird oder es eine Gebühr gibt für schnelleres Reparieren. Diese beiden Ausnahmen, können aber auch nicht berücksichtigt werden durch den Algorithmus, weshalb das allererste Verfahren, vermutlich das beste algorithmische Verfahren darstellt, da jeder fair an der Reihe ist, in der Reihenfolge wann dessen Auftrag eingegangen ist.

3 Lösungsidee

Der Aufgabenstellung zu entnehmen, ist das jeder Auftrag eine Dauer und einen Zeitpunkt besitzt, wo der Auftrag in die Fahrradwerkstatt eingeht. Außerdem werden die Aufträge in den Eingabedateien nach Eingangszeitpunkt sortiert, wo der erste Auftrag den Auftrag darstellt, der zuerst eingeht. Die Aufträge dürfen aber immer nur von 9 Uhr bis 17 Uhr (also dem Arbeitstag) abgearbeitet werden und erst nachdem ein Auftrag nach dessen Dauer fertig bearbeitet wurde, darf der nächste bearbeitet werden.

Obwohl es unterschiedliche Verfahren gibt, wie der nächst zu bearbeitende Auftrag ausgewählt wird, kann trotzdem ein verfahrensunabhängiger Algorithmus die Auswahl des nächsten Auftrages delegieren und sich selber mit der Simulation der Fahrradwerkstatt beschäftigen. Denn immer muss der nächst

zu bearbeitende Auftrag einer der schon eingegangen Aufträge sein und gibt es keinen, muss gewartet werden, egal welches Verfahren zur Auswahl des zu bearbeitenden Auftrages benutzt wird. Dieser verfahrensunabhängige Algorithmus wird im folgenden zuerst beschrieben.

Der Algorithmus funktioniert so, dass sich immer die Zeit gemerkt wird und in einer Schleife immer ein Auftrag bearbeitet wird und dementsprechend die Zeit immer weiter addiert wird. Wenn es aber keinen Auftrag zu bearbeiten gibt, wird die Zeit auf den nächsten Zeitpunkt gesetzt, wo es wieder mindestens einen Auftrag zur Wahl gibt, der eingegangen ist und bearbeitet werden kann.

Im konkreten wird sich als erstes gemerkt, was die aktuelle Zeit ist und wann der jetzige Arbeitstag zu Ende ist. Am Anfang ist die aktuelle Zeit 9 Uhr und der Arbeitstag zu Ende um 17 Uhr. Die maximale und gesamte Wartezeit werden sich auch gemerkt. Alle Aufträge und ob sie schon bearbeitet worden sind, werden sich auch gemerkt.

In einer äußeren Schleife wird zunächst ein Auftrag ausgewählt, nach dem momentanen Verfahren zur Auswahl der Aufträge. Dieser Auftrag muss nach der aktuellen Zeit schon eingegangen sein, darf aber noch nicht bearbeitet worden sein, da er zum Bearbeiten ausgewählt wird.

Man merkt sich nun die schon bearbeitete Zeit des Auftrages (also am Anfang 0). Der Auftrag wird dann mit dem folgenden Algorithmus bearbeitet: In einer Schleife, bis der Auftrag bearbeitet wurde, also bis die schon bearbeitete Zeit des Auftrages, der Dauer des Auftrages entspricht:

- Wenn die Zeit am gemerkten Ende des jetzigen Arbeitstages ist, wird die Zeit zum nächsten Tag um 9 Uhr gestellt, also dem beginn des nächsten Arbeitstages und es wird sich nun das Ende des nächsten Arbeitstages gemerkt, also am nächsten Tag um 17 Uhr. D.h. die Simulation befindet vorher am Ende eines Arbeitstages und befindet sich nun am Anfang des nächsten Arbeitstages und weiß wann dieser Endet.
- Anhand der Differenz zwischen der momentanen Zeit und des gemerkten Endes dieses Arbeitstages, kann ermittelt werden wie viel Zeit noch in in diesem Arbeitstag verbleibt.
- Anhand der Differenz zwischen dem Fortschritt des Auftrages und der Dauer des Auftrages, kann ermittelt werden wie lange der Auftrag noch bearbeitet werden muss.
- Das kürzere der beiden, also entweder Verbleib des Arbeitstages oder Verbleib des Auftrages, wird gewählt. Diese Zeit wird dann an die aktuelle Zeit und die bearbeitete Zeit des Auftrages addiert.
- So befindet man sich also nun zeitlich am Ende der Bearbeitung des Auftrages, oder am Ende dieses Tages, wenn der Auftrag nicht mehr im aktuellen Tag zu schaffen war. Es gibt auch einen Sonderfall, wenn der Auftrag genau am Ende eines Tages bearbeitet wurde. Dann wird aber trotzdem, da der Auftrag bearbeitet ist, die Schleife abgebrochen. Ist aber nur des Ende des Tages erreicht, wird beim nächsten Durchlauf der Schleife zum nächsten Tag gesprungen und dort weiter bearbeitet, wie es im ersten Stichpunkt beschrieben ist.

Der Algorithmus unterbricht also immer am Ende des Tages den Auftrag, um dann am nächsten Tag weiter den Auftrag zu bearbeiten. Nach der Schleife ist der Algorithmus also an dem Zeitpunkt, wo die Aufgabe fertig bearbeitet wurde.

Da zu jedem Auftrag die Eingangszeit vermerkt ist, kann anhand der Differenz der Eingangszeit und der aktuellen Zeit, die Wartezeit des Kunden für den ausgewählten Auftrag errechnet werden. Diese wird der gesamten Wartezeit addiert und wenn sie größer ist als das bisherige Maximum an Wartezeit, wird das bisherige Maximum damit ersetzt.

Nun kann der nächste Auftrag in der äußeren Schleife bearbeitet werden. Wenn jedoch der Sonderfall eintritt, dass der Auftrag genau dann beendet wurde, wo der Tag endet, kann jedoch nicht direkt der nächste Auftrag zum Bearbeiten ausgewählt werden. Denn die aktuelle Zeit wäre noch am Ende eines Tages, was heißt, dass nur von den Aufträgen ausgewählt werden kann, die bis dahin eingegangen sind. Da jedoch erst am nächsten Tag angefangen wird, am nächsten Auftrag zu arbeiten, könnte noch ein Auftrag am Anfang dieses Tages eingehen, der auch zur Auswahl stehen sollte. Deswegen muss im Sonderfall vor dem nächsten Durchgang in der äußeren Schleife die Zeit gleich dem Anfang des nächsten Tages gesetzt werden und sich der Anfang des nächsten Tages gemerkt wird. Dann befindet man sich vor der Auswahl des nächsten Auftrages auch am nächsten Tag.

Wenn in der äußeren Schleife aber kein nicht-bearbeiteter Auftrag ausgewählt werden kann, muss gewartet werden, bis mindestens ein nächste Auftrag eingegangen ist. Es kann jedoch nicht einfach die aktuelle Zeit direkt auf die Eingangszeit des nächsten Auftrages gesetzt werden, da sich das Ende des aktuellen Arbeitstages immer gemerkt werden muss. Deswegen wird in einer Schleife bis die aktuelle Zeit an dem Zeitpunkt ist, wo der nächste Auftrag eingeht oder schon eingegangen ist, folgendes gemacht:

- Die aktuelle Zeit wird an den Zeitpunkt gesetzt, wo der nächste Auftrag eingeht.
- Ist die aktuelle Zeit gleich oder nach dem Ende des aktuellen Arbeitstages, wird die Zeit an den Anfang des nächsten Arbeitstages gesetzt und das gemerkte Ende des Arbeitstages an das Ende des nächsten Arbeitstages.

Die äußere Schleife wird beendet, wenn alle Aufträge bearbeitet worden sind. Für die Aufgabenstellung wurde die maximale Wartezeit bereits gemerkt. Die durchschnittliche Wartezeit kann aus der gesamten Wartezeit geteilt durch die Anzahl aller Aufträge ermittelt werden.

Damit im Algorithmus sich gemerkt werden kann, welche Aufträge eingegangen sind beziehungsweise bearbeitet wurden und welche noch nicht, werden zwei Container betrachtet, einen für die eingegangen und einen für die noch nicht eingegangenen. Ist ein Auftrag bearbeitet, ist er in keinem der beiden Container. Am Anfang der äußeren Schleife, wo ein Auftrag zu bearbeiten ausgewählt wird, werden vorher alle Aufträge, die nach der aktuellen Zeit eingegangen sind, aber noch im Container der noch nicht eingegangen Aufträge sind, in den Container der schon eingegangenen Aufträge bewegt. Es wird dann von diesem Container ein Auftrag zum Bearbeiten ausgewählt und entfernt. Sind beide Container also leer, sind alle Aufträge bearbeitet worden und die äußere Schleife kann dann beendet werden.

Was bei der Delegation des Verfahrens zur Bestimmung des nächsten Auftrages beachtet werden muss, ist das wenn in der äußeren Schleife ein neuer Auftrag in den Container der eingegangenen Aufträge bewegt wird, dieser einsortiert wird. Das heißt das der Container der eingegangen Aufträge sortiert ist und zwar Verfahrens abhängig, da das einsortieren auch an das Verfahren delegiert wird.

Was also unterschiedlich bei den Verfahren sein kann, ist wie ein neu eingegangener Auftrag in den schon eingegangenen Aufträgen einsortiert wird, aber auch wie vom sortierten Container der schon eingegangenen Aufträgen ein zu bearbeitender Auftrag ermittelt wird.

Die beiden in der Aufgabenstellung verlangten Verfahren der Bestimmung des nächst zu bearbeitenden Auftrages wählen immer den ersten Auftrag aus dem sortierten Container der eingegangenen Aufträge. Sie sortieren jedoch die Aufträge unterschiedlich. Das ursprüngliche Verfahren hängt einfach immer nur den neusten eingegangen Auftrag in den Container der schon eingegangenen Aufträge hinten ein. Da beim Auswählen des zu bearbeitenden Auftrages das erste ausgewählt wird, ist dies das was von den eingegangen Aufträgen als erstes eingegangen ist und werden damit in der Reihenfolge ihres Eingangs, wie Marc es plant, ausgewählt. Das neue Verfahren sortiert neue eingehende Aufträge nach Dauer ein, so dass das erste immer der Auftrag mit der geringsten Dauer ist.

4 Umsetzung

Bei der Umsetzung wird Zeit, beziehungsweise Dauer, als Integer in Minuten dargestellt und Zeitpunkte als Zeit von 0 aus. Wenn der gemerkte Zeitpunkt für das Ende des jetzigen Arbeitstages (17 Uhr), zum Ende des nächsten Arbeitstages (17 Uhr am nächsten Tag) werden soll, muss der Zeitpunkt lediglich mit 24 Stunden ($24 * 60$ Minuten) addiert werden. Wenn die aktuelle Zeit gleich dem Ende des jetzigen Arbeitstages ist (17 Uhr) und auf den Anfang des nächsten Arbeitstages (nächsten Tag 9 Uhr) gestellt werden soll, muss die Zeit mit $(24 - 17) + 9 = 16$ Stunden ($16 * 60$ Minuten) addiert werden Wenn die Zeit größer als das gemerkten Ende des Arbeitstages ist, man aber die Zeit nur auf dem Anfang des Arbeitstages nach den gemerkten Ende das Arbeitstages setzten möchte, muss man die Zeit gleich des jetzigen gemerkten Ende des Arbeitstages +16 setzen.

Ein Auftrag wird als Klassenobjekt `Task` mit Dauer und Eingangszeitpunkt dargestellt.

Die nicht eingegangenen Aufträge werden dargestellt als eine Liste von allen Aufträgen, wo jedoch sich gemerkt wird, welche Aufträge davon noch nicht eingegangen sind. Da alle Aufträge sortiert sind, nach Eingangszeitpunkt, wird sich konkret nur der Index gemerkt, wo der zugehörige Auftrag der erste Auftrag ist, der noch nicht eingegangen ist. Wenn am Anfang der äußeren Schleife alle Aufträge, die sich in den eingegangen Aufträgen befinden sollten, dorthin bewegt werden, wird lediglich der Index verschoben, bis der Auftrag vom Index nicht mehr eingegangen sein sollte. Also bis der Auftrag vom Index wieder der erste Auftrag ist, der nach der aktuellen Zeit nicht mehr eingegangen ist. Während also der Index verschoben wird, kann immer der Auftrag vom Index in die eingegangen Aufträge eingetragen werden, bis ein Auftrag gefunden wird, der nach der aktuellen Zeit noch nicht eingegangen ist und deshalb nicht eingetragen wird. Die schon eingegangenen Aufträge werden dargestellt als sortierte Liste von Aufträgen, die nur die Aufträge enthält, die auch schon eingegangen sind.

Die Delegation der Bestimmung des nächst zu bearbeitenden Auftrages erfolgt durch eine abstrakte Klasse `TaskPriorityDelegate`. An einem Objekt einer Unterklasse dieser abstrakten Klassen wird also das Auswählen des nächsten Auftrages delegiert. Jedes Verfahren des Auswählen des nächsten

Auftrages wird also als eine Unterklasse dieser abstrakten Klasse dargestellt. Die abstrakte Klasse hat zwei Methoden, die sich jeweils um das Sortieren der neu eingegangenen Aufträge in die schon eingegangen Aufträge und um das Auswählen der schon sortierten eingegangen Aufträge kümmern. Bei der sortierenden Methode, wird die Liste an schon eingegangenen Aufträgen gegeben und ein gegebener Auftrag soll dort einsortiert werden. Bei der auswählenden Methode wird auch die Liste an schon eingegangenen Aufträgen gegeben und es muss ein Auftrag daraus entfernt werden und zurückgegeben werden. Da beide Verfahren und damit Implementationen der Klasse immer den ersten Auftrag auswählen und sich nur in der Sortierung unterscheiden ist die Auswahl des Auftrages schon implementiert in der abstrakten Klasse und muss nicht mehr in den Unterklassen unterschieden. Da aber andere Verfahren möglich sind, die entweder nicht sortieren oder die Sortierung nur teilweise berücksichtigen, gibt es diese Methode trotzdem.

Die Klasse `FiFoTaskPriorityDelegate` implementiert `TaskPriorityDelegate` nach der Reihenfolge der eingehenden Aufträge. Die Klasse `ShortestDurationTaskPriorityDelegate` implementiert `TaskPriorityDelegate` nach der aufsteigenden Dauer der eingehenden Aufträge. In der Methode `simulateProcessingTasks` wird die Abarbeitung von den Aufträgen simuliert. Es wird ein `TaskPriorityDelegate` gegeben und eine Liste von Aufträgen. Die Methode delegiert also das Sortieren und Auswählen von eingegangenen Aufträgen an den `TaskPriorityDelegate`, indem die beiden Methoden des gegebenen Objektes Aufgerufen wird mit der Liste, wo einsortiert werden soll und gegebenenfalls dem Auftrag der einsortiert werden soll. Wenn ein Auftrag ausgewählt werden soll, gibt die aufgerufene Methode diesen zurück. Ausgegeben wird dann die durchschnittliche, längste und gesamte Wartezeit. Die Methode `simulateProcessingTasks` wird deshalb in der `main` Methode mit jeder Kombination von jedem `TaskPriorityDelegate` und jeder Eingabedatei aufgerufen. Alle Unterklassen vom `TaskPriorityDelegate` sind als Array in der Konstante `taskPriorityDelegates` eingetragen.

5 Beispiele

5.1 Erstes Beispiel

Vergleich zwischen Auswahl von Aufträgen nach ihrer Eingangsreihenfolge und Auswahl nach kürzester Dauer:

Eingangszeit (In Stunden | Minuten seit t0)	**Dauer (In Stunden | Minuten)**
9 | 540	2 | 120
11 | 600	2 | 180
10 | 660	3 | 120
x 13 | 780	5 | 300

Table 1: Erstes Beispiel Aufträge

Eingangsreihenfolge:

```
*Äußerer Schleifendurchgang*
Aktuelle Zeit: 540
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 540, dauer: 120}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 540, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 660

    Aufgabe Abgeschlossen um 660
    Die Aufgabe ging ein um 540
    Es musste also 120 Minuten auf die Aufgabe gewartet werden

*Äußerer Schleifendurchgang*
```

[...]

```
*Äußerer Schleifendurchgang*
Aktuelle Zeit: 840
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 660, dauer: 120}
{eingangsZeit: 780, dauer: 300}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 660, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 960

    Aufgabe Abgeschlossen um 960
    Die Aufgabe ging ein um 660
    Es musste also 300 Minuten auf die Aufgabe gewartet werden


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 960
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 780, dauer: 300}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 780, dauer: 300}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 60 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 1020

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Man befindet sich am Ende des Tages, neuer Tag bei 1980
    Neues Ende des Tages bei 2460
    Es wurden 240 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 2220

    Aufgabe Abgeschlossen um 2220
    Die Aufgabe ging ein um 780
    Es musste also 1440 Minuten auf die Aufgabe gewartet werden


Durchschnittliche Wartezeit pro Auftrag: 525.0 minuten
Gesamte Wartezeit für alle Aufträge: 2100 minuten
Längste Wartezeit in allen Aufträgen: 1440 minuten
```

Man sieht hier wie Schritt für Schritt jede Aufgabe bearbeitet wird. Die ersten drei Aufgaben werden am gleichen Tag erledigt, doch die vierte Aufgabe dauert bis zum Tag danach, weshalb man zwei innere

Schleifengänge sehen kann. Der erste innere Schleifengang bearbeitet die Aufgabe bis zum Ende des Tages, während die zweite bis zum Ende der Aufgabe bearbeitet am nächsten Tag.

Auswahl nach kürzester Dauer:

```
*Äußerer Schleifendurchgang*


[...]


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 660
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 660, dauer: 120}
{eingangsZeit: 600, dauer: 180}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 660, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 780

    Aufgabe Abgeschlossen um 780
    Die Aufgabe ging ein um 660
    Es musste also 120 Minuten auf die Aufgabe gewartet werden


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 780
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 600, dauer: 180}
{eingangsZeit: 780, dauer: 300}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 600, dauer: 180}

     *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
     Es wurden 180 der Aufgabe abgearbeitet
     Man ist nun bei der Zeit 960

     Aufgabe Abgeschlossen um 960
     Die Aufgabe ging ein um 600
     Es musste also 360 Minuten auf die Aufgabe gewartet werden


*Äußerer Schleifendurchgang*


[...]


Durchschnittliche Wartezeit pro Auftrag: 510.0 minuten
Gesamte Wartezeit für alle Aufträge: 2040 minuten
Längste Wartezeit in allen Aufträgen: 1440 minuten
```

Im Vergleich kann man erkennen, dass die zweite und dritte Aufgabe bei beiden anders gewählt wird, da es beim zweiten äußeren Schleifendurchgang dazu kommt, dass zwischen eingegangenen Aufgaben gewählt werden kann. Einer dieser Aufgaben ist früher in die Liste gekommen und die andere Dauert dafür nicht so lange. Bei dem Verfahren der Auswahl nach kürzester Dauer wird das mit der kürzeren Dauer genommen und beim Verfahren die von der Eingangsreihenfolge abhängig ist, wird die Aufgabe genommen die zuerst eingegangen ist.

5.2 Zweites Beispiel

Beachtung des Sonderfalls, wo ein Auftrag genau dann endet, wo der Arbeitstag endet. Dabei ist die Auswahl der Aufträge nach der kürzesten Dauer wichtig, da als der Arbeitstag endet es einen Auftrag gibt der aber länger ist, als einer der eingeht am Anfang des nächsten Tages. Es kann also nur der zweite

Auftrag ausgewählt wird, wenn nachdem ein Auftrag genau am Ende eines Arbeitstages endet, bevor der Auswahl des nächsten Auftrages die aktuelle Zeit der nächste Tag ist. Ansonsten wird der längere Auftrag ausgewählt, welches falsch wäre.

Eingangszeit (In Stunden | Minuten seit t0)	**Dauer (In Stunden | Minuten)**
9 | 540	8 | 480
17 | 1020	3 | 180
x 33 | 1980	2 | 120

Table 2: Zweites Beispiel Aufträge

Auswahl nach kürzester Dauer::

```
*Äußerer Schleifendurchgang*
Aktuelle Zeit: 540
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 540, dauer: 480}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 540, dauer: 480}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 480 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 1020

    Aufgabe Abgeschlossen um 1020
    Die Aufgabe ging ein um 540
    Es musste also 480 Minuten auf die Aufgabe gewartet werden
    Sonderfall: Die Aufgabe wurde am Ende des Tages fertiggestellt und der nächste Tag beginnt nun


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 1980
Ende dieses Arbeitstages: 2460
Alle schon eingegangenen Aufgaben
{eingangsZeit: 1980, dauer: 120}
{eingangsZeit: 1020, dauer: 180}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 1980, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 2100

    Aufgabe Abgeschlossen um 2100
    Die Aufgabe ging ein um 1980
    Es musste also 120 Minuten auf die Aufgabe gewartet werden


*Äußerer Schleifendurchgang*


[...]


Durchschnittliche Wartezeit pro Auftrag: 620.0 minuten
Gesamte Wartezeit für alle Aufträge: 1860 minuten
Längste Wartezeit in allen Aufträgen: 1260 minuten
```

Man sieht im ersten äußeren Schleifendurchgang, wie eine 8 Stündige Aufgabe genau den ganzen Tag füllt und deshalb am Ende die aktuelle Zeit zum nächsten Tag setzt. Im nächsten Schleifen Durchgang, wird dann der Auftrag genommen der um 1980 eingeht, also die nun aktuelle Zeit. Wäre die Zeit also nicht zum nächsten Tag gesetzt worden, wäre dieser Auftrag nicht genommen.

5.3 Drittes Beispiel

Pausen zwischen Aufgaben: Wenn es keine eingegangene Aufgaben gibt, muss gewartet werden bis die nächsten eintrifft.

Eingangszeit (In Stunden | Minuten seit t0)	**Dauer (In Stunden | Minuten)**
9 | 540	2 | 120
x 13 | 780	2 | 120

Table 3: Drittes Beispiel Aufträge

Eingangsreihenfolge:

```
*Äußerer Schleifendurchgang*
Aktuelle Zeit: 540
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 540, dauer: 120}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 540, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 660

    Aufgabe Abgeschlossen um 660
    Die Aufgabe ging ein um 540
    Es musste also 120 Minuten auf die Aufgabe gewartet werden


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 660
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben

Es steht keine Aufgabe zur Auswahl, dementsprechend muss bis 780 gewartet werden
Zeit ist abgearbeitet


*Äußerer Schleifendurchgang*
Aktuelle Zeit: 780
Ende dieses Arbeitstages: 1020
Alle schon eingegangenen Aufgaben
{eingangsZeit: 780, dauer: 120}

Es gibt eingegangene Aufgaben, Ausgewählte Aufgabe: {eingangsZeit: 780, dauer: 120}

    *Innerer Schleifendurchgang um Aufgabe zu bearbeiten*
    Es wurden 120 der Aufgabe abgearbeitet
    Man ist nun bei der Zeit 900

    Aufgabe Abgeschlossen um 900
    Die Aufgabe ging ein um 780
    Es musste also 120 Minuten auf die Aufgabe gewartet werden


Durchschnittliche Wartezeit pro Auftrag: 120.0 minuten
Gesamte Wartezeit für alle Aufträge: 240 minuten
Längste Wartezeit in allen Aufträgen: 120 minuten
```

Man sieht in der Ausgabe, das zwischen den beiden Aufträgen zwei Stunden gewartet werden muss. Dann fängt der dritte äußere Schleifendurchgang genau da an, bis gewartet wurde und es gibt mindestens eine eingegangene Aufgabe, die ausgewählt und bearbeitet werden kann.

6 Quellcode

```java
/**
 * Ein Auftrag
 *
 * @param entranceTime Der Eingangszeitpunkt des Auftrages in Minuten von t0
 * @param duration     Die Dauer des Auftrages in Minuten
 */
private record Task(int entranceTime, int duration) {
}
```

Klasse Task

```java
/**
 * Eine Klasse die entscheidet in welcher Reihenfolge die eingehenden Aufträge verwaltet werden sollen.
 */
abstract private static class TaskPriorityDelegate {
    /**
     * @return Der Name dieses Verfahrens, in welcher Reihenfolge die eingehenden Aufträge verwaltet werden sollen.
     */
    public abstract String getName();

    /**
     * Wo in der aktuellen Liste der schon eingegangen Aufträge ein neuer Auftrag eingehen soll.
     * Das hat aber keinen direkten Einfluss darauf, wann dieser neue Auftrag angenommen werden soll,
     * denn pickTask entscheidet dass.
     * Durch eine Sortierung der Liste durch sortTaskIntoCurrentTaskList
     * kann aber pickTask oft effizienter vorgehen.
     * <p>
     * Die implementierte Methode soll also an einer stellte in der taskQueue den newTask einsortieren.
     *
     * @param taskQueue Die aktuelle Liste der eingegangenen Aufträge
     * @param newTask   Der neue Auftrag, der in die Liste, der eingegangen Aufträge eingenommen werden soll.
     */
    public abstract void sortTaskIntoCurrentTaskList(List<Task> taskQueue, Task newTask);

    /**
     * Wählt den nächsten Auftrag zum Bearbeiten von der Liste der schon eingegangen Aufträge aus,
     * entfernt ihn aus dieser Liste und gibt ihn zurück.
     * Durch Sortieren dieser Liste in sortTaskIntoCurrentTaskList ist ein effizienteres Auswählen möglich.
     * <p>
     * Standardmäßig nimmt, wählt die Methode den ersten Auftrag in der Liste aus, da davon ausgegangen wird,
     * dass meistens die Liste in sortTaskIntoCurrentTaskList sortiert wird.
     *
     * @param taskQueue Die Liste der eingegangenen Aufträge.
     * @return Der Auftrag, der als nächstes bearbeitet werden soll.
     */
    public Task pickTask(List<Task> taskQueue) {
        return taskQueue.remove(0);
    }
}
```

Klasse TaskPriorityDelegate

```
    /**
     * Eine Implementation von {@link TaskPriorityDelegate}, die immer den Auftrag der
    als neuestes eingegangen ist,
     * als letztes bearbeitet.
     */
    static class FiFoTaskPriorityDelegate extends TaskPriorityDelegate {
        @Override
        public String getName() {
            return "Auftrag Reihenfolge priorisieren";
        }

        /**
         * Sortiert immer den neuen Auftrag immer ganz hinten. Da wenn der Auftrag in
    pickTask ausgewählt wird,
         * immer der erste genommen wird, wird damit der älteste ausgewählt.
         */
        @Override
        public void sortTaskIntoCurrentTaskList(List<Task> taskQueue, Task newTask) {
            taskQueue.add(newTask);
        }
    }

```

Klasse FiFoTaskPriorityDelegate

```
  /**
     * Eine Implementation von {@link TaskPriorityDelegate}, der immer den Auftrag als nä
    chstes bearbeitet,
     * der am kürzesten dauert.
     */
    static class ShortestDurationTaskPriorityDelegate extends TaskPriorityDelegate {
        @Override
        public String getName() {
            return "Kürzester Auftrag priorisieren";
        }

        /**
         * Sortiert den neuen Auftrag in die Liste, der eingegangen Aufträge, ein, nach
    absteigender Dauer.
         * Da wenn der Auftrag in pickTask ausgewählt wird, immer der kürzeste genommen
    wird.
         */
        @Override
        public void sortTaskIntoCurrentTaskList(List<Task> taskQueue, Task newTask) {
            for (int i = 0; i < taskQueue.size(); i++) {
                if (taskQueue.get(i).duration > newTask.duration) {
                    taskQueue.add(i, newTask);
                    return;
                }
            }
            taskQueue.add(newTask);
        }
    }

```

Klasse ShortestDurationTaskPriorityDelegate

```
  /**
     * Alle Verfahren zur Auswahl des nächst zu bearbeitenden Auftrages.
     */
     private static final TaskPriorityDelegate[] taskPriorityDelegates = new
    TaskPriorityDelegate[]{
            new FiFoTaskPriorityDelegate(),
            new ShortestDurationTaskPriorityDelegate(),
      };

```

Konstante taskPriorityDelegates

```java
/**
   * Für alle Eingabedateien werden alle Verfahren durchgeführt
   * mit {@link Main#simulateProcessingTasks(List, TaskPriorityDelegate)}.
   */
private static void simulateProcessingTasks(List<Task> tasks, TaskPriorityDelegate
  taskPriorityDelegate) {
      int maxWaitedTime = 0;
      int allWaitingTime = 0;
      int time = 9 * 60;
      int nextBreak = 17 * 60;
      int firstTaskNotOnTaskQueue = 0;
      List<Task> taskQueue = new ArrayList<>(tasks.size());
      while (firstTaskNotOnTaskQueue != tasks.size() || !taskQueue.isEmpty()) {
          while (firstTaskNotOnTaskQueue != tasks.size() && tasks.get(
  firstTaskNotOnTaskQueue).entranceTime <= time) {
              taskPriorityDelegate.sortTaskIntoCurrentTaskList(taskQueue, tasks.get(
  firstTaskNotOnTaskQueue));
              firstTaskNotOnTaskQueue++;
          }
          if (!taskQueue.isEmpty()) {
              Task currentlyExecutingTask = taskPriorityDelegate.pickTask(taskQueue);
              int currentlyExecutingTaskProgress = 0;
              while (currentlyExecutingTaskProgress != currentlyExecutingTask.duration)
   {
                  assert currentlyExecutingTaskProgress < currentlyExecutingTask.
  duration;
                  assert time <= nextBreak;
                  if (time == nextBreak) {
                      time += (9 + (24 - 17)) * 60;
                      nextBreak += 24 * 60;
                  }
                  int passedTime = currentlyExecutingTask.duration -
  currentlyExecutingTaskProgress;
                  if (time + passedTime > nextBreak) {
                      passedTime = nextBreak - time;
                  }
                  time += passedTime;
                  currentlyExecutingTaskProgress += passedTime;
              }
              int waitedTime = time - currentlyExecutingTask.entranceTime;
              allWaitingTime += waitedTime;
              maxWaitedTime = Math.max(waitedTime, maxWaitedTime);
              if (time == nextBreak) {
                  time += (9 + (24 - 17)) * 60;
                  nextBreak += 24 * 60;
              }
          } else {
              int targetedTime = tasks.get(firstTaskNotOnTaskQueue).entranceTime;
              while (time < targetedTime) {
                  time = targetedTime;
                  if (time >= nextBreak) {
                      time = nextBreak + (9 + (24 - 17)) * 60;
                      nextBreak += 24 * 60;
                  }
              }
          }
      }
      double averageTaskProcessingTime = (double) allWaitingTime / (double) tasks.size
  ();
      averageTaskProcessingTime = ((double) Math.round(averageTaskProcessingTime * 10))
   / 10;

      System.out.println("Durchschnittliche Wartezeit pro Auftrag: " +
  averageTaskProcessingTime + " minuten");
      System.out.println("Gesamte Wartezeit für alle Aufträge: " + allWaitingTime + "
  minuten");
      System.out.println("Längste Wartezeit in allen Aufträgen: " + maxWaitedTime + "
  minuten");
  }

```

Methode simulateProcessingTasks