Stefan Waidele
Ensisheimer Straße 2
79395 Neuenburg am Rhein
Stefan@Waidele.info

AKAD University
Immatrikulationsnummer: 102 81 71

Assignment im Modul JAV02

Erstellung einer Multithread—Anwendung zur Untersuchung von Erzeuger–Verbraucher–Szenarien

Betreuer: Prof. Dr. Franz–Karl Schmatzer

3. März 2015

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

FIFO	First In, First Out
LIFO	Last In, First Out
Mutex	Mutually Exclusive Lock
E	Erzeuger
V	Verbraucher
Q	Queue
P	Produkt
EPK	Ereignisgesteuerte Prozesskette
UML	Unified Modelling Language

Quellcodeverzeichnis

1 Einleitung

1.1 Begründung der Problemstellung

Erzeuger–Verbraucher–Konstellationen sind sowohl in der Wirtschaft als auch in der Informationsverarbeitung sehr häufig zu beobachten und prägen viele alltäglichen Vorgänge. Aufgrund der Vielzahl der zu betrachtenden Parameter, die neben der Anzahl der Erzeuger und Verbraucher auch die jeweilige Produktions- bzw. Verbrauchsgeschwindigkeit und die Lagerkapazität für fertige Erzeugnisse einschließt handelt es sich hierbei trotz ihrer Alltäglichkeit um komplexe Systeme, welche mich rein oberflächlichen Untersuchungen nicht komplett erfasst werden können.

Des Weiteren sind Erzeuger–Verbraucher–Szenarien auch in der Informatik anzutreffen. Dies gilt sowohl für die Systemebene[1] als auch auf der Anwendungsebene[2]

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, eine Erzeuger–Verbraucher Anwendung in Java zu entwickeln. Anhand dieser Anwendung sollen die typischen Problemstellungen nebenläufiger Anwendungen diskutiert, sowie das Laufzeitverhalten der erstellten Anwendung beobachtet und erörtert werden.

Die zu erstellende Anwendung soll als konkretes Beispiel eine Pizzeria simulieren, die laufend Pizzas herstellt und für Kunden zur Abholung bereit hält. Mehrere Verbraucher entnehmen in zufälligen Intervallen Pizzas aus diesem Vorrat. Hierbei werden sowohl der Erzeuger als auch jeder Verbraucher in einem jeweils eigenen Threads simuliert. Es ist darauf zu achten, dass kein Kunde eine Pizza aus einem leeren Vorratsspeicher entnehmen kann. Ebenfalls kann der Pizzabäcker keine

[1] z.B. Logging oder Interrupts
[2] z.B. die Darstellung von Messdaten oder der Abruf von Netzwerkresourcen

weitere Pizza in einen bereits vollen Vorratsspeicher hinzufügen. In diesen Fällen sollen die Akteure warten bis ein Produkt bzw. ein freier Platz zur Verfügung steht. Sowohl beim Einstellen als bei der Abholung soll zur Dokumentation des Laufzeitverhaltens Erzeuger (E) bzw. Verbraucher (V) sowie die in der Warteschlange bzw. Queue (Q) befindliche Anzahl der Pizzen bzw. Produkte (P) ausgegeben werden.

Zunächst werden hierzu in den Kapitel *2 Erzeuger und Verbraucher* und *3 Genutzte Sprachmerkmale von Java* durch Literaturrecherche die Grundlagen der Problemstellung sowie die notwendigen Werkzeuge der Programmiersprache herausgearbeitet. Anschließend wird in Kapitel *5 Implementierung* das Simulationsprogramm erstellt. Die unterschiedlichen Simulationsläufe liefern die Daten für die Untersuchung des Laufzeitverhaltens in Kapitel *6 Laufzeitbetrachtungen.*

1.3 Abgrenzungen

Die zur Implementierung der Threadsynchronisation zumindest indirekt genutzten Konstrukte `synchronized()` und `ReentrantLock` sowie die als Warteschlange eingesetzte `LinkedBlockingQueue` werden nicht grundlegend erklärt sondern nur in dem Ausmaß beschrieben, wie es für diese Arbeit notwendig ist. Ausführlichere Erklärungen finden sich in den angegebenen Quellen.

In der Aufgabenstellung wird gefordert, dass „wenn ein Erzeuger bzw. Verbraucher die Queue betritt“ der Füllstand der Queue ausgegeben werden soll. Für diese Arbeit wird angenommen, dass diese Ausgabe unmittelbar vor Anforderung des kritischen Abschnitts erfolgt und somit nicht Teil desselben ist. Eine Implementierung der Ausgabe innerhalb des kritischen Abschnitts wäre mit den hier besprochenen Mitteln problemlos möglich, würde jedoch durch die dann notwendigen Maßnahmen zur Vermeidung von Deadlocks eine sinnvolle gemeinsame Basisklasse für `Erzeuger` und `Verbraucher` ausschließen. Da sich die Ausgabe hierdurch nur in

Einzelfällen[3] und geringen Details[4] ändern würde, wird einem sauberen objektorientierten Entwurf den Vorzug gegeben. In Abschnitt *2.4 Korrektes Logging* werden Lösungsmöglichkeiten für dieses Problem genannt, deren Implementierung jedoch den Umfang dieser Arbeit überschreiten würde.

Die in Abschnitt *2.5 Freiwilliges und unfreiwilliges Warten* dargestellte getrennte Erfassung von freiwilligen und unfreiwilligen Wartezeiten wird in dieser Arbeit nicht durchgeführt, da die Bewertung dieser unterschiedlichen Zeiten stark vom konkreten Einsatzgebiet abhängt.

[3] Falls ein `Akteur` zwischen der Ausgabe der Statuszeile und dem Entnehmen des Elements von der Queue von einem anderen `Akteur` unterbrochen wird

[4] In diesem Fall würde der ein `Akteur` einen falschen Füllstand der Queue ausgeben. Obwohl dies genau die Art von Fehler ist, die in dieser Arbeit besprochen wird, wird dies vom Autor in Kauf genommen, da es sich lediglich um die Logging–Ausgabe und nicht um die Grundfunktionalität handelt.

2 Erzeuger und Verbraucher

2.1 Erzeuger–Verbraucher—Muster

Bei Erzeuger–Verbraucher–Konstellationen treten Produzenten und Konsumenten miteinander über eine Warteschlange bzw. einen Buffer miteinander in Kontakt. Jeder der Akteure handelt hierbei unabhängig von den anderen und quasi gleichzeitig.[5]

Hierbei ist es unerheblich, was hierbei erzeugt bzw. verbraucht wird. Es kann sich um reale Güter handeln, die von Kunden nachgefragt werden, oder auch um Informationen, die Abgerufen werden. Ebenfalls ist es unerheblich, ob der Erzeuger tatsächlich den Bedarf des Verbrauchers befriedigt, oder ob Anfragen erzeugt werden, die dann vom Verbraucher abgearbeitet bzw. gelöst werden.[6] Auch können die Anzahl der Erzeuger und die der Verbraucher beliebig gewählt werden oder sogar schwanken. Allen Situationen ist jedoch gemeinsam, dass das Erzeugnis nur einem Verbraucher zur Verfügung steht und nur ein mal konsumiert werden kann.

Durch das einfügen eines Zwischenspeichers stellen Erzeuger–Verbraucher–Konstellationen einen Mechanismus dar, durch den eine beliebige Anzahl von Produzenten und Verbraucher miteinander kommunizieren können. Es handelt es sich somit um ein Entuwrfsmuster, welches in der Programmierung von nebenläufigen Anwendungen.[7]

2.2 Zwischenspeicher

Bei der Zwischenspeicher (engl. Buffer) können verschiedene Prinzipien zum Einsatz kommen. Bei Last In, First Out (LIFO) werden diejenigen Elemente

[5] vgl. Tanenbaum & Woodhull (2005), S. 76f
[6] Ein Beispiel hierfür ist z.B. ein Trouble–Ticket–System: Hier werden vom Nutzer Supportanfragen „erzeugt“, die dann von den Supportmitarbeitern „verbraucht” werden.
[7] vgl. Hoffmann & Lienhart (2008), S. 111

zuerst entnommen, die als letztes hinzugefügt wurden. Dieses Verfahren kommt bei der Implementierung mithilfe eines Kellerspeichers (engl. Stack) zum Tragen. Das First In, First Out (FIFO) Prinzip, bei dem das zuerst hinzugefügte Element auch als erstes wieder entnommen wird kann durch eine Warteschlange (engl. Queue) oder mithilfe eines Ringbuffers implementiert werden.[8]

Bei der im Rahmen dieser Arbeit wird eine FIFO–Warteschlange (engl. Queue) genutzt.[9]

2.3 Kritische Abschnitte

Damit Verbraucher nicht auf eine leere Queue zugreift, muss zunächst geprüft werden, ob ein Element zur Verfügung steht. Da Prüfung und Zugriff jedoch getrennte Operationen darstellen, besteht die Möglichkeit, dass ein weiterer Verbraucher das letzte Element zwischen diesen beiden Vorgängen entnimmt.[10] Um solche konkurrierende Zugriffe zu Verhindern, muss verhindert werden, dass sich mehrere Verbraucher gleichzeitig in diesem kritischen Abschnitt befinden.

Eine solche Zugangskontrolle lässt im sich Programm mithilfe von Mutex' bzw. Semaphoren realisieren. Beiden Konzepten ist gemein, dass ein Thread vor dem Eintritt in einen kritischen Abschnitt dessen Verfügbarkeit prüft und gegebenenfalls solange wartet, bis diese eintritt. Anschließend wird vermerkt, dass sich ein Thread im kritischen Abschnitt befindet. Beim Verlassen wird diese Markierung wieder entfernt. Bei einem Mutex wird die Zugangskontrolle über einen Wahrheitswert geregelt, weshalb hier maximal ein Thread den kritischen Bereich betreten kann.[11] Bei Semaphoren kommt ein Zähler zum Einsatz, über den die Anzahl der gleichzeitig erlaubten Threads gesteuert werden kann.[12] Hierbei ist sicher zu stellen, dass zwischen der Verfügbarkeitsprüfung und der Sperre für andere

[8] vgl. Sedgewick & Wayne (2011), Abschnitt 1.3.3.8 und 1.3.3.9
[9] vgl. Oracle (Hrsg.) (1993b)
[10] Bei mehreren Erzeugern tritt das Problem analog bei fast voller Warteschlange auf.
[11] vgl. Silberschatz et al. (2012), Abschnitt 5.5
[12] vgl. Silberschatz et al. (2012), Abschnitt 5.6

Threads kein anderer Thread diesen kritischen Abschnitt betreten kann.[13] stattfindet. Code, der kritische Abschnitte korrekt behandelt wird auch als threadsicher bezeichnet.

Hierbei ist zu beachten, dass sich Verbraucherthreads und Erzeugerthreads nicht gegenseitig blockieren. Bei FIFO Warteschlangen dürfen diese Operationen auch quasi gleichzeitig ablaufen. Hier können Verbraucher das älteste Element der Warteschlange entnehmen, während ein Erzeuger ein neues Element hinzufügt. Es bestehen somit zwei getrennte Kritische Abschnitte[14], die unabhängig voneinander überwacht werden können.

Im Falle einer Realisierung mit Hilfe einer LIFO Warteschlange besteht ein gemeinsamer kritischer Abschnitt. Erzeuger und Verbraucher können hier nicht gleichzeitig auf die Warteschlange zugreifen. Hierbei ist sicher zu stellen, dass die Akteure bei leerer bzw. voller Warteschlange den kritischen Abschnitt wieder verlassen, um den Programmfluss nicht dauerhaft in einem sogenannten „Deadlock“ zu blockieren.[15]

Ein weiterer kritischer Abschnitt der mit einem Mutex geschützt werden muss besteht bei der Ausgabe der in der Aufgabenstellung geforderten Logging–Informationen auf den Bildschirm. Hierbei könnte ein Thread von einem weiteren während der Ausgabe unterbrochen werden. Dieser zweite Thread würde dann seine Informationen zwischen die des Ersten schreiben. Um die Bildschirmausgabe konkurrieren alle Verbraucher und der Erzeuger.

[13] Diese Operation muss nicht atomar im strengen Sinne sein. Kontextwechsel sind durchaus zu erlauben, jedoch nicht zu Threads, die um die betreffende Resource konkurrieren.

[14] „Prüfung ob Element vorhanden und Entnahme“ sowie „Prüfung ob Platz vorhanden und Einstellen”

[15] Ein blockierender Verbraucher bei leerer Queue würde hierbei auch einen Erzeuger blockieren, der das erwartete Element somit nicht liefern kann.

2.4 Korrektes Logging

Wie bereits in Abschnitt *1.3 Abgrenzungen* erwähnt, kann ein Akteure `A1` zwischen der Bildschirmausgabe und dem eigentlichen Zugriff auf die Warteschlange von einem anderen Akteur `A2` unterbrochen werden. Hierbei wird die Ausgabe von `A1` inkorrekt.

Dies kann dadurch vermieden werden, dass die Bildschirmausgabe ebenfalls in den kritischen Bereich mit aufgenommen wird. Hierzu ist statt den unabhängigen Locks für das Hinzufügen und das Entnehmen von Elementen ein Gemeinsames notwendig, welches einen Kontextwechsel zwischen Ausgabe und Zugriff verhindert. Hierdurch können sich bei ganz voller oder ganz leerer Warteschlange wartende Erzeuger und Verbraucher gegenseitig so blockieren, dass es zu einer Deadlocksituation kommt. Dies könnte vermieden werden, wenn Verbraucher bei leerer Warteschlange den kritischen Abschnitt direkt wieder verlassen, ohne ein Element zu entnehmen bzw. Erzeuger bei voller Warteschlange das aktuelle Element verwerfen, um den kritischen Abschnitt wieder verlassen zu können.

Eine weitere Möglichkeit, ist eine zweistufigen Ausgabe unter Nutzung eines Ticket–Systems: Ein Akteur betritt den kritischen Abschnitt des Ticket–Systems[16], erzeugt die Ausgabezeile, speichert diese lokal und erhält ein „Ticket“ in Form einer fortlaufenden Nummer bevor er diesen ersten kritischen Abschnitt verlässt. Anschließend betritt der Akteur den kritischen Abschnitt der Warteschlange[17] und prüft die Verfügbarkeit von Elementen bzw. Platz in der Warteschlange. Gegebenenfalls wird gewartet. Dann übermittelt der Akteur die Ticketnummer an das Ticketsystem. Hierdurch werden alle älteren Tickets ungültig. Ist das übergebene Ticket gültig, wird die Bildschirmausgabe durchgeführt, auf die Warteschlange zugegriffen und der kritische Abschnitt verlassen. Ist das übergebene Ticket ungültig, wird der kritische Abschnitt direkt verlassen, um den Vorgang wird ohne Wartezeit neu begonnen.

[16] Der kritische Abschnitt des Ticket–Systems ist für Erzeuger und Verbraucher gemeinsam.
[17] Dieser ist wie im Abschnitt *2.3 Kritische Abschnitte* beschrieben für Erzeuger und Verbraucher unabhängig voneinander.

2.5 Freiwilliges und unfreiwilliges Warten

Wie im Abschnitt *5.1 Klasse: Akteur* noch näher beschrieben wird, besteht sowohl der Vorgang des Erzeugens als auch der des Verbrauchens im wesentlichen aus drei Phasen: Die Zeit, die für die Herstellung bzw. für den Konsum eines Elements benötigt wird[18], welche von außen als freiwillige Wartezeit wahrgenommen wird; die Wartezeit, die gegebenenfalls beim Warten auf den kritischen Abschnitt anfällt und somit als unfreiwillig anzusehen ist, sowie dem eigentlichen Zugriff auf die Warteschlange. Bei Untersuchungen konkreter Erzeuger–Verbraucher–Systeme können diese Zeiten entsprechend gemessen und unterschiedlich bewertet werden.

[18] Dies schließt die Phase der „Sättigung“ mit ein.

3 Genutzte Sprachmerkmale von Java

3.1 Collections, generische Klassen, Iteratoren

Die zu erstellende Anwendung benötigt Klassen, in denen mehrere Objekte[19] gespeichert werden können. Die Java–Klassenbibliothek stellt hierzu verschiedene Containerklassen zur Verfügung, welche von der Klasse `Collection` abstammen[20] und Objekte beliebigen Typs aufnehmen.[21]

Um dennoch eine Prüfung der Typen zu ermöglichen, wurde in Java ab der Version 5.0 das Konzept der generischen Programmierung (auch „parametrisierte Typen“) eingeführt. Hierbei kann bei der Instanziierung angegeben werden, welcher Klasse die in der Collection zu speichernden Objekte angehören werden. So wird durch die Deklaration `ArrayList <Verbraucher> v = new ArrayList<Verbraucher>();` ein Behälter vom Typ `ArrayList` erzeugt, der Elemente vom Typ `Verbraucher` speichert.[22]

Iteratoren stellen eine Möglichkeit dar, auf die in einer Collection gespeicherten Elemente zuzugreifen. Hierbei ist keine Kenntnis der tatsächlichen Stelle, an der das jeweilige Element gespeichert ist notwendig. Sind alle Elemente abgearbeitet, liefert die Methode `hasNext()` als Ergebnis `FALSE`.[23] In der zu erstellenden Anwendung wird ein Iterator implizit in Form einer erweiterten `for`–Schleife genutzt.

3.2 Threads

Die im Abschnitt *2.1 Erzeuger–Verbraucher—Muster* beschriebene Unabhängigkeit und Gleichzeitigkeit der Akteure lässt sich in Java mit Hilfe von Threads implementieren. Hierbei handelt es sich um Anweisungsstränge die innerhalb

[19] Hierbei handelt es sich um die Verbraucherthread–Objekte sowie um die Elemente der Warteschlange
[20] Ausnahme: `java.util.Map`, vgl. Heinisch et al. (2011), Seite 688
[21] vgl. Oracle (Hrsg.) (1993a)
[22] vgl. Heinisch et al. (2011), Seite 622ff
[23] vgl. Heinisch et al. (2011), Seite 692

eines Prozesses nebenläufig abgearbeitet werden.[24] Hierzu ist eine Ableitung der Klasse `Thread` zu erstellen, welche die Methode `run()` implementiert. Bei der Instanziierung der Objekte werden dann die entsprechenden Threads erzeugt, welche anschließend durch den Aufruf der Methode `start()` gestartet werden.[25]

3.3 Threadsicherheit

Die Java–Klassenbibliothek bietet keine Semaphoren zur Sychronisierung von Threads an. Ein entsprechender Mechanismus müsste somit in einer eigenen Klasse implementiert werden. Hierbei ist besonders zu beachten, dass wie in Abschnitt *2.3 Kritische Abschnitte* beschrieben die Verfügbarkeitsprüfung und die Sperre der anderen Threads in einer quasi–atomaren Operation erfolgt. Dies kann mit Hilfe des Schlüsselwortes `synchronized` sichergestellt werden, welches eine Mutually Exclusive Lock (Mutex) um den damit markierten Block bzw. um die damit ausgezeichnete Methode legt.[26] Alternativ können über die Klasse `ReentrantLock` von Codeblöcken unabhängige Sperren angefordert werden.[27] Die Überprüfung der Queue auf vorhandene Elemente bzw. Platz zum Einfügen neuer Elemente muss hierbei genau so wie der eigentliche Zugriff sind hierbei ebenfalls zu kodieren.

Die generische Klasse `LinkedBlockingQueue<E>` implementiert alle im vorhergehenden Absatz genannten Aufgaben. Hierbei handelt es sich um eine FIFO–Queue, die threadsicheren Zugriff auf die Elemente ermöglicht. Beim Hinzufügen durch die Methode `put()` wird gegebenenfalls gewartet, bis Platz vorhanden ist, beim Entnehmen durch `take` wird gegebenenfalls gewartet, bis die Queue nicht mehr leer ist.[28] Intern verwendet die Klasse `ReentrantLock` zur Vermeidung von konkurrierenden Zugriffen[29]

[24] vgl. Heinisch et al. (2011), Seite 748f
[25] vgl. Heinisch et al. (2011), Seite 752f
[26] vgl. Heinisch et al. (2011), Seite 767
[27] vgl. Oracle (Hrsg.) (1993d)
[28] vgl. Oracle (Hrsg.) (1993b)
[29] vgl. Lea (b), Zeile 377–402

Die Methode `System.out.println()` ist durch interne Verwendung `synchronized()` `threadsicher.`[30] Hierdurch ist auch der Zugriff auf die Bildschirmausgabe für den Fall geregelt, dass mehrere Threads gleichzeitig die Methode `log()` der Klasse `Logger` aufrufen.

3.4 Zufällige Wartezeiten

Pseudo–Zufallszahlen werden in Java mithilfe von Objekten der Klasse Random generiert. Die Methode `nextInt(n)` liefert eine zufällige Ganzzahl r für die gilt $0 \leq r < n$. Durch die Addition von 1 kann hierdurch eine Wartezeit von mindestens einer Zeiteinheit erzeugt werden.

Gewartet wird innerhalb eines Thread mit der Methode `sleep()`, welche Wartezeit in Millisekunden als Argument erwartet. Daher kann die Zeiteinheit sehr klein gewählt werden, was einen schnellen Lauf der Simulation ermöglicht. Oder durch einen entsprechenden Faktor kann die Zeiteinheit größer gewählt werden, damit der Verlauf der Simulation besser live beobachtet werden kann.

[30] vgl. Oracle (Hrsg.) (1993c), Zeile 771—776

4 Modellierung

4.1 Ereignisgesteuerte Prozessketten

Aus der Aufgabenbeschreibung ergeben sich die in Abbildung 2 dargestellten Ereignisgesteuerte Prozesskette (EPK) für Erzeuger und Verbraucher. Hierbei ist zu erkennen, dass sich die Prozesse bei identischem Ablaufschema lediglich in den durchgeführten Prüfungen und Operationen unterscheiden. Hierdurch liegt es nahe, dass die Ablaufsteuerung im objektorientierten Entwurf in einer abstrakten Basisklasse modelliert wird, und nur die unterschiedlichen Handlungen in den jeweiligen spezialisierten Ableitungen konkretisiert werden.

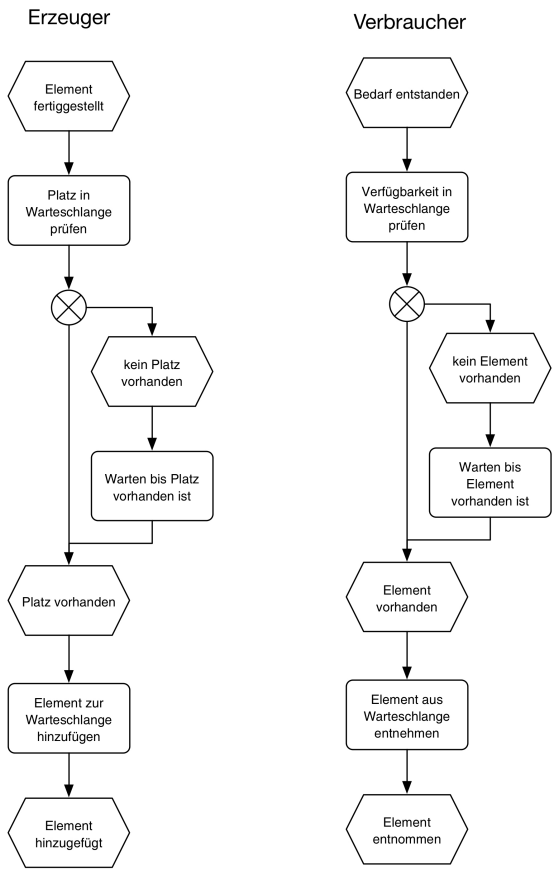

Abb. 1: Ereignisgesteuerte Prozessketten

4.2 UML–Diagramm

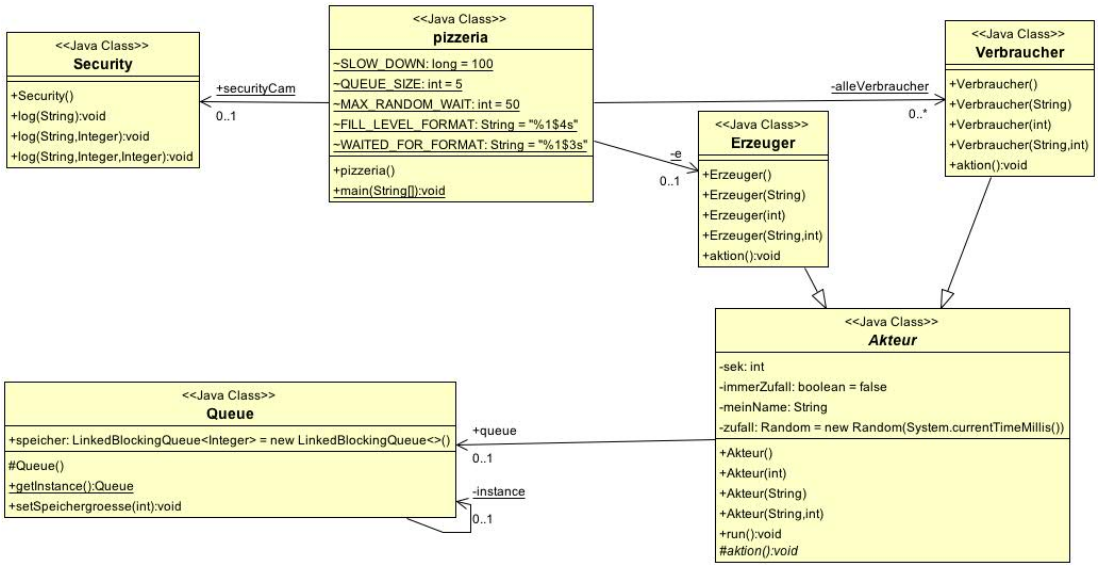

Abb. 2: UML–Diagramm

5 Implementierung

5.1 Klasse: Akteur

In der der abstrakten Klasse `Akteur` werden die Eigenschaften und Methoden zusammengefasst, die den Klassen `Erzeuger` und `Verbraucher` gemeinsam sind. Dazu gehören die Angaben zur Wartezeit `sek` bzw. `immerZufall` sowie die Bezeichnung `meinName`, die bei der Ausgabe dazu dienen kann, die Akteure zu unterscheiden. Diese Variablen werden von den Konstruktoren beim Erzeugen der Objekte befüllt. Des weiteren wird ein Generator für Pseude–Zufallszahlen sowie eine Referenz auf die Warteschlange benötigt.

In der Methode `run()`, die beim Starten des Threads aufgerufen wird, wird zunächst wie gefordert der Name und der Füllstand der Queue, ergänzt mit der Wartezeit ausgegeben. Anschließend wird die abstrakte Methode `aktion()` aufgerufen, welche die spezifischen Aktionen der abzuleitenden Klassen ausführt. Abschließend wird durch Aufruf der Methode `sleep()` der Elternklasse `Thread` die vorgegebene oder zufällig bestimmte Zeit gewartet.

Die Klasse `Akteur` implementiert keinen Zugriff auf die Queue und betritt somit keinen kritischen Abschnitt.

5.2 Klasse: Erzeuger

Die von `Akteur` abgeleitete Klasse `Erzeuger` implementiert lediglich die Methode `aktion()`. Hierzu muss der kritische Abschnitt betreten werden. Hierin muss geprüft werden, ob Platz in der Queue vorhanden ist. Falls nicht, muss gewartet werden, bis dies der Fall ist. Wenn Platz vohanden ist, wird das produzierte Element[31] in der Queue abgelegt. Anschließend wird der kritische Abschnitt wieder verlassen.

[31] Hierbei handelt es sich in der Simulation um den Integerwert 1

Diese Vorgehensweise entspricht der Implementierung der `put()` der Klasse `LinkedBlockingQueue`, welche indirekt über die Klasse `Queue` aufgerufen wird.[32]

Wie im Abschnitt *2.3 Kritische Abschnitte* ausgeführt, sind das Hinzufügen und das Entnehmen von Elementen bei FIFO Warteschlangen getrennte kritische Abschnitte. Somit kann das Warten im Falle einer vollen Warteschlange durchgeführt werden, ohne eine Deadlock–Situation herbeizuführen.

5.3 Klasse: Verbraucher

Die von `Akteur` abgeleitete Klasse `Verbraucher` implementiert lediglich die Methode `aktion()`. Hierzu muss der kritische Abschnitt betreten werden. Hierin muss geprüft werden, ob mindestens ein Element in der Queue vorhanden ist. Falls nicht, muss gewartet werden, bis dies der Fall ist. Wenn ein Element vohanden ist, wird das älteste entnommen. Anschließend wird der kritische Abschnitt wieder verlassen.

Diese Vorgehensweise entspricht der Implementierung der `take()` der Klasse `LinkedBlockingQueue` welche indirekt über die Klasse `Queue` aufgerufen wird.[33]

5.4 Sonstige Programmmerkmale

Die Klasse `Queue` realisiert das Singleton–Entwurfsmuster, um ein Element der Klasse `LinkedBlockingQueue` der gewünschten Größe allen beteiligten Objekten und damit allen Threads zugänglich zu machen.[34]

Die Klasse `Logger` übernimmt die geforderte Ausgabe von „E“ bzw. „V” gefolgt von der Angabe der Zahl der momentan in der Warteschlange vorhandenen Elementen. Ergänzt werden diese Angaben von der Wartezeit[35] und der

[32] vgl. Lea (a)
[33] vgl. Lea (b)
[34] vgl. Gamma et al. (1994), Seite 127ff
[35] Hierbei ist die Wartezeit innerhalb des kritischen Abschnitts nicht berücksichtigt

dem Verbraucher zugewiesenen laufenden Nummer. Durch die Ausgabe des kompletten, zuvor zusammengesetzten Ausgabestrings in einem einzigen Aufruf von `System.out.println()` wird hier der konkurrierende Zugriff auf die gemeinsame Resource der Bildschirmausgabe threadischer geregelt.[36]

Die Klasse Pizzeria stellt das Gerüst der Anwendung dar. Hierin werden neben einigen Konstanten auch die benötigten Obkekte der Klassen `Logger` und `Erzeuger` sowie eine `ArrayList` für die nach dem Programmstart zu bestimmende Anzahl der Verbraucher definiert. Die Wartezeiten für die Verbraucher und den Erzeuger werden abgefragt und die Threads erzeugt und gestartet. Anschließend werden alle weiteren Schritte des Programms in den Threads ausgeführt.

[36] siehe auch Abschnitt *3.3 Threadsicherheit*

6 Laufzeitbetrachtungen

6.1 Generelle Beobachtungen

6.2 Erzeuger schneller als Verbraucher

6.3 Erzeuger langsamer als Verbraucher

6.4 Ezeuger und Verbraucher gleich schnell

7 Fazit & Ausblick

7.1 Fazit

7.2 Ausblick

Ausbau der Anwendung, um verschiedene Szenarien simulieren zu können, z.B. variables Arbeitstempo der Erzeuger (höhere Belastung von Maschinen oder Menschen), zusätzlichen Erzeugern (Wie lange dauert das Anfahren?), Hinterlegung von Kosten für Lagerhaltung, Wartezeiten bei Erzeuger oder Verbraucher, etc. Bis hin zu hinterlegter Kalkulation, das den Endpreis des Produktes ermittelt und somit steuernd auf die Nachfrage einwirkt. Hierdurch könnten Aussagen zu Stoßzeiten (Gastronomie, Einzelhandel, Energie, Webserver, ...) getroffen werden.

Für eine solche feingliedrige Simulation ist eine eigene Implementierung der Queue oder evt. sogar des Threading denkbar, um an beliebigen Stellen den Zustand des Systems abfegen und analysieren zu können.

A Quelltext der Anwendung

A.1 Klasse: Pizzeria

```java
import java.util.Scanner;
import java.util.ArrayList;

public class Pizzeria {

  static final long SLOW_DOWN = 1;
  static final int QUEUE_SIZE = 15;
  static final int MAX_RANDOM_WAIT = 50;
  static final String FILL_LEVEL_FORMAT = "%1$4s";
  static final String WAITED_FOR_FORMAT = "%1$3s";

  public static Logger logger = new Logger();

  private static Erzeuger e;
  private static ArrayList <Verbraucher> alleVerbraucher
  = new ArrayList<Verbraucher>();

  public static void main(String[] args) {
    int nv;    // Anzahl der Verbraucher

    Scanner s = new Scanner(System.in);
    System.out.print("Wartezeit für Erzeuger? "
        + "(0 für jedes Mal eine neue zufällige Wartezeit) ");
    e = new Erzeuger("E ", s.nextInt());

    System.out.print("Wie viele Verbraucher? ");
    nv = s.nextInt();
    for (int i=1; i<=nv; i++) {
      System.out.print("Wartezeit für Verbraucher " + i + "? "
          + "(0 für jedes Mal eine neue zufällige Wartezeit) ");
      alleVerbraucher.add(new Verbraucher("V"+i, s.nextInt()));
    }
    s.close();

    System.out.println("Pizzeria JAVO2 hat geöffnet.");

    Queue queue = Queue.getInstance();
    queue.setSpeichergroesse(QUEUE_SIZE);

    e.start();
    for(Verbraucher verbraucher: alleVerbraucher) {
      verbraucher.start();
    }
  }
}
```

Quellcode 1: Pizzeria.java

A.2 Klasse: Akteur

```java
import java.util.Random;

public abstract class Akteur extends Thread {
  // Wartezeit zwischen dem Abholen
  private int sek;
  private boolean immerZufall = false;
  private String meinName;

  public Queue queue = Queue.getInstance();

  // Pseudo-Zufallszahlengenerator
  private Random zufall = new Random(System.currentTimeMillis());

  // Default-Constructor: Keine Parameter, zufällige Wartezeit,
  // Name "A" für Akteur
  public Akteur() {
    sek = 1 + zufall.nextInt(Pizzeria.MAX_RANDOM_WAIT-1);
    meinName = "A";
  }
  // Constructor: Vorgegebene Wartezeit & Name
  public Akteur(String s, int i) {
    sek = i;
    if(i==0) {
      immerZufall = true;
    }
    meinName = s;
  }

  public void run() {
    while (true) {
      // Wartezeit ermitteln...
      // Falls jedes Mal zufällig gewartet werden soll:
      if (immerZufall) {
        // Zufällige Wartezeit
        sek = 1 + zufall.nextInt(Pizzeria.MAX_RANDOM_WAIT-1);
      }
      // Warten... (bzw. satt sein, oder nächstes Element vorbereiten, ..)
      try { Thread.sleep(sek*Pizzeria.SLOW_DOWN);}
      catch (InterruptedException e) {};

      // Loggen...
      Pizzeria.logger.log(meinName, sek, queue.speicher.size());

      // Handeln!
      aktion();

    }
  }

  protected abstract void aktion();
}
```

Quellcode 2: Akteur.java

A.3 Klasse: Erzeuger

```java
public class Erzeuger extends Akteur {

  // Default-Constructor: Keine Parameter, zufällige Wartezeit
  public Erzeuger() {
    super();
  }
  // Constructor: Vorgegebene Wartezeit & Name
  public Erzeuger(String s, int i) {
    super(s, i);
  }

  public void aktion() {
    try{queue.speicher.put(1);} catch(InterruptedException e) {};
  }
}
```

Quellcode 3: Erzeuger.java

A.4 Klasse: Verbraucher

```java
public class Verbraucher extends Akteur {

  // Default-Constructor: Keine Parameter, zufällige Wartezeit
  public Verbraucher() {
    super();
  }
  // Constructor: Vorgegebene Wartezeit & Name
  public Verbraucher(String s, int i) {
    super(s, i);
  }

  public void aktion() {
      try { queue.speicher.take();} catch(InterruptedException e) {};
  }
}
```

Quellcode 4: Verbraucher.java

A.5 Klasse: Queue

```
1  import java.util.concurrent.LinkedBlockingQueue;
2
3  public class Queue {
4    // Klassisches Singleton Entwurfsmuster
5    // vgl. GoF, Seite 127ff
6    // Java Implementierung z.B.
7    // http://www.javaworld.com/article/2073352/core-java/simply-singleton.html
8    private static Queue instance = null;
9    public   LinkedBlockingQueue<Integer> speicher
10   = new LinkedBlockingQueue<>();
11
12   // Leerer, geschützter Constructor verhindert Instanziierung
13   protected Queue() {}
14
15   public static Queue getInstance() {
16     if(instance == null) {
17       instance = new Queue();
18     }
19     return instance;
20   }
21   // Möglichkeit die Größe der Queue zu bestimmen/verändern
22   public void setSpeichergroesse(int i) {
23     // Neue Queue mit gewünschter Größe erzeugen & referenzieren
24     speicher = new LinkedBlockingQueue<>(i);
25     // Die alte Queue wird nicht mehr referenziert
26     // und per Garbage Collection freigegeben,
27   }
28 }
```

Quellcode 5: Queue.java

A.6 Klasse: Logger

```java
public class Logger {
  public void log(String s, Integer t, Integer n) {
    // Zuerst die Zeile erzeugen...
    s = s + "(" + String.format(Pizzeria.WAITED_FOR_FORMAT, t) + "sek): "
          + String.format(Pizzeria.FILL_LEVEL_FORMAT, n) + " |";
    // n Sternchen
    for (int i=0;i<n;i++) {
      s = s + "*";
    }
    // und QUEUE_SIZE-n Leerzeichen
    for (int i=n;i<Pizzeria.QUEUE_SIZE;i++){
      s = s + " ";
    }
    s = s + "|";
    //... und anschließend mit EINEM println() ausgeben
    // (wg. Threadsicherheit)
    System.out.println(s);
  }
}
```

Quellcode 6: Logger.java

B Ergebnisse verschiedenener Programmläufe

B.1 Erzeuger schneller als Verbraucher

```
Wartezeit für Erzeuger? (0 für jedes Mal eine neue zufällige Wartezeit) 1
Wie viele Verbraucher? 3
Wartezeit für Verbraucher 1? (0 für jedes Mal eine neue zufällige Wartezeit) 3
Wartezeit für Verbraucher 2? (0 für jedes Mal eine neue zufällige Wartezeit) 4
Wartezeit für Verbraucher 3? (0 für jedes Mal eine neue zufällige Wartezeit) 12
Pizzeria JAVO2 hat geöffnet.
V1(  3sek):    0 |              |
V3( 12sek):    0 |              |
E (  1sek):    0 |              |
V2(  4sek):    0 |              |
E (  1sek):    0 |              |
E (  1sek):    0 |              |
E (  1sek):    0 |              |
V2(  4sek):    1 |*             |
E (  1sek):    0 |              |
V1(  3sek):    1 |*             |
E (  1sek):    0 |              |
E (  1sek):    1 |*             |
E (  1sek):    2 |**            |
V2(  4sek):    3 |***           |
V1(  3sek):    2 |**            |
E (  1sek):    1 |*             |
E (  1sek):    2 |**            |
E (  1sek):    3 |***           |
V1(  3sek):    4 |****          |
E (  1sek):    3 |***           |
V2(  4sek):    4 |****          |
E (  1sek):    3 |***           |
V3( 12sek):    4 |****          |
E (  1sek):    3 |***           |
V1(  3sek):    4 |****          |
E (  1sek):    3 |***           |
E (  1sek):    4 |****          |
V2(  4sek):    5 |*****         |
E (  1sek):    4 |****          |
V1(  3sek):    5 |*****         |
E (  1sek):    4 |****          |
E (  1sek):    5 |*****         |
E (  1sek):    6 |******        |
V2(  4sek):    7 |*******       |
V1(  3sek):    6 |******        |
E (  1sek):    5 |*****         |
E (  1sek):    6 |******        |
E (  1sek):    7 |*******       |
V1(  3sek):    8 |********      |
E (  1sek):    7 |*******       |
V2(  4sek):    8 |********      |
E (  1sek):    7 |*******       |
V3( 12sek):    8 |********      |
E (  1sek):    7 |*******       |
V1(  3sek):    8 |********      |
E (  1sek):    7 |*******       |
E (  1sek):    8 |********      |
V2(  4sek):    9 |*********     |
E (  1sek):    8 |********      |
V1(  3sek):    9 |*********     |
E (  1sek):    8 |********      |
E (  1sek):    9 |*********     |
V2(  4sek):   10 |**********    |
E (  1sek):    9 |*********     |
V1(  3sek):   10 |**********    |
E (  1sek):    9 |*********     |
E (  1sek):   10 |**********    |
V1(  3sek):   11 |***********   |
E (  1sek):   10 |**********    |
V2(  4sek):   11 |***********   |
E (  1sek):   10 |**********    |
V3( 12sek):   11 |***********   |
E (  1sek):   10 |**********    |
V1(  3sek):   11 |***********   |
E (  1sek):   10 |**********    |
E (  1sek):   11 |***********   |
V2(  4sek):   12 |************  |
E (  1sek):   11 |***********   |
V1(  3sek):   12 |************  |
E (  1sek):   11 |***********   |
E (  1sek):   12 |************  |
E (  1sek):   13 |************* |
V2(  4sek):   14 |**************|
V1(  3sek):   13 |************* |
E (  1sek):   12 |************  |
E (  1sek):   13 |************* |
E (  1sek):   14 |**************|
V1(  3sek):   15 |**************|
E (  1sek):   14 |**************|
V2(  4sek):   15 |**************|
E (  1sek):   14 |**************|
V3( 12sek):   15 |**************|
E (  1sek):   14 |**************|
V1(  3sek):   15 |**************|
E (  1sek):   14 |**************|
E (  1sek):   15 |**************|
V2(  4sek):   15 |**************|
E (  1sek):   15 |**************|
V1(  3sek):   15 |**************|
E (  1sek):   15 |**************|
```

Literatur– und Quellenverzeichnis

Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John (1994): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Heinisch, Cornelia, Müller-Hoffmann, Frank & Goll, Joachim (2011): *Java als erste Programmiersprache*. 6. Aufl. Vieweg + Teubner Verlag, Wiesbaden.

Hoffmann, Simon & Lienhart, Rainer (2008): *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*. Springer-Verlag.

Lea, Doug (a): *OpenJDK Sourcecode: java.util.concurrent.LinkedBlockingQueue.put()*. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/LinkedBlockingQueue.java#LinkedBlockingQueue.take%28%29`, abgerufen am 22.02.2015.

Lea, Doug (b): *OpenJDK Sourcecode: java.util.concurrent.LinkedBlockingQueue.take()*. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/LinkedBlockingQueue.java#LinkedBlockingQueue.take%28%29`, abgerufen am 18.02.2015.

Oracle (Hrsg.) (1993a): *Java Documentation: Class Collection<E>*. `http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`, abgerufen am 18.02.2015.

Oracle (Hrsg.) (1993b): *Java Documentation: Class LinkedBlockingQueue<E>*. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html`, abgerufen am 14.02.2015.

Oracle (Hrsg.) (1993c): *Java Documentation: Class PrintStream*. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/io/PrintStream.java#PrintStream.println%28%29`, abgerufen am 26.02.2015.

Oracle (Hrsg.) (1993d): *Java Documentation: Class ReentrantLock*. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html`, abgerufen am 18.02.2015.

Sedgewick, Robert & Wayne, Kevin (2011): *Algorithms*. 4th edition, video enhanced Aufl. Addison-Wesley Professional.

Silberschatz, Abraham, Galvin, Peter B. & Gagne, Greg (2012): *Operating System Concepts, 9th Edition*. E–Book. John Wiley & Sons.

Tanenbaum, Andrew S. & Woodhull, Albert S. (2005): *Operating Systems Design and Implementation*. 3rd Aufl. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Eidesstattliche Erklärung

Ich versichere, dass ich das beiliegende Assignment selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe.

(Datum, Ort) (Unterschrift)