DIPLOMARBEIT

zum Thema

Entwicklung eines Testdatengenerator für Market Basket Analysis für e-commerce Anwendungen

ausgeführt am

Institut für Informationsverarbeitung und -wirtschaft

Abteilung für Informationswirtschaft

der Wirtschaftsuniversität Wien

Begutachter: o. Univ. Prof. Dkfm. Dr. Wolfgang Janko

Betreuer: Dr. Michael Hahsler

Eingereicht von

Martin Groblschegg

9150630

Wien, im Jänner 2003

Inhaltsverzeichnis

0 Einleitung 4
0.1 Aufgabenstellung und Zielsetzung der Arbeit 4
0.2 Aufbau und Struktur der Arbeit 5
1 Die Grundlagen des Data Mining 7
1.1 Knowledge Discovery in Databases 7
1.2 Data Mining Verfahren 8
1.2.1 Grundlagen der Assoziationsregeln 8
1.2.2 Generischer Algorithmus 9
1.2.3 Arten von Assoziationsregeln 10
2 IBM's Almaden - Testdatengenerator 13
2.1 Almaden 13
2.2 Arbeitsweise des Generators 13
3 Ehrenbergs Repeat Buying Theorie als Basis zur Entwicklung des Testgenerators 17
4 Berechnung der Transaktionen 23
4.1 Ausgangsdaten 23
4.1.1 Produkte 23
4.1.2 Kunden 23
4.1.3 Zeitraum 23
4.2 Kundenverhalten 24
4.2.1 Kaufzeitpunkte 24
4.2.2 Bildung der Transaktionen 25
5 Algorithmen zur Findung von Zufallszahlen 27
5.1 Die benötigten Zufallszahlenverteilungen 27
5.1.1 Exponentialverteilung 27
5.1.2 Gammaverteilung 28
5.2 Zufallszahlengeneratoren 28
5.2.1 Echte Zufallszahlengeneratoren 28
5.2.2 Pseudozufallszahlengeneratoren 29
5.2.3 Umrechnung in andere Verteilungen 35
6 Dokumentation des Programms 39
6.1 Kompilieren des Programms 39

6.2 Benutzerdokumentation 39
6.2.1 Startparameter 39
6.2.2 Bemerkungen 40
6.3 Systemdokumentation 41
6.3.1 Überblick über das Systemdesign 41
6.3.2 Implementierung im Detail 59
6.3.3 Beschreibung der Ausgabedateien 59
6.3.4 Ansätze für Programmerweiterungen 61
7 Auswertung der erzeugten Testdaten 63
7.1 Detailbetrachtung einiger Produkte 65
8 Sourcecode 68
8.1 Datei: TDG.h 68
8.2 Datei: TDG.cpp 72
9 Abbildungsverzeichnis 87
10 Literaturverzeichnis 89

0 Einleitung

0.1 Aufgabenstellung und Zielsetzung der Arbeit

Im heutigen E-Commerce sind Recommender Systeme nicht mehr wegzudenken. Da dem Käufer im Online-Bereich kein Shopping-Gefühl wie im Geschäft geboten werden kann, muss dies durch EDV-Systeme ausgeglichen werden. In herkömmlichen Geschäften wird intensiv mit einer gezielten Platzierung von Produkten gearbeitet. Das bekannteste Beispiel sind die Artikel die vor einer Supermarktkassa den in der Schlange wartenden Kunden angeboten werden. Diese Möglichkeit des zusätzlichen Umsatzes gibt es in Online-Geschäften nicht und wird daher durch andere Möglichkeiten, wie zum Beispiel Recommender Systeme kompensiert.

Recommender Systeme bestimmen an Hand von Assoziationsregeln zu einem bestimmten Artikel jene Produkte, die von vielen anderen Benutzern gleichzeitig mit dem Artikel gekauft werden. Somit ist das System in der Lage dem Käufer zu seiner Wahl passende Produkte anzubieten.

Heute werden diese Systeme bereits von bekannten Online-Händlern wie Barnes & Noble oder Amazon.com eingesetzt und werden in der Zukunft immer weitere Verbreitung finden [GeHa02].

Diese Assoziationsregeln werden aus dem Datenbestand der Transaktionen errechnet, die die Anwendung gesammelt hat. Transaktionsdaten enthalten den Kunden und die Artikel die der Kunde in bei einem Einkauf erworben hat. Der Kassabon, den man an einer Supermarktkasse erhält kann man sich als einen Transaktionsdatensatz vorstellen. Je besser diese Regeln sind, desto genauer können passende Produkte vorgeschlagen werden.

Für die Entwicklung neuer Algorithmen, ist der Zugang zu großen Datenmengen notwendig. Diese Daten werden jedoch von Handelskonzernen nicht immer bereitgestellt. Außerdem sind echte Datenbestände meist durch kurzfristige Werbeaktionen und Preisnachlässe verfälscht. Eine Preisaktion kann zu einer temporären Veränderung des Kaufverhaltens der Kunden führen. Aus diesen

Gründen ist es notwendig für die Arbeit mit den Algorithmen eigene Datenbestände zu erzeugen, die die Daten nach bestimmten Regeln erstellen und keine Verfälschung beinhalten.

Ein bekannter Vertreter der Testdatengeneratoren ist „Quest Synthetic Data Generation Code“ von Quest, der Datamining Group von IBM. [Quest02]

Dieser Testdatengenerator erzeugt Daten, die konventionellen Handelsbetrieben entsprechen sollen. Wie jedoch Untersuchungen in [zhkoma01] zeigen, funktionieren bestehende Algorithmen für E-Commerce Anwendung nicht mit den Datenbeständen, die vom IBM-Generator erstellt werden. Ein Grund dafür ist, dass die Verteilung der Transaktionsgröße (die Anzahl der gekauften Produkte in einer Transaktion) nicht den Daten aus Webshops entsprechen. In Webshops haben die meisten Transaktionen eine Größe von 1 wobei die Daten aus dem Generator meistens eine Größe von 10 haben. Näheres zu der Untersuchung der Datenbestände in [zhkoma01].

Ziel dieser Arbeit ist es einen Testdatengenerator zu erstellen, der Test-Daten generiert, die für E-Commerce Lösungen relevant sind. Entsprechend sind die Transaktionsdaten in diesem Generator nach der Theorie von Ehrenberg NB-verteilt.

0.2 Aufbau und Struktur der Arbeit

Zu Beginn der Arbeit werden in Kapitel 1 die Grundlagen des Data Mining dargelegt wobei besonderes Augenmerk auf die Assoziationsregeln gelegt wird. Darauf aufbauend wird der Testdatengenerator von IBM der Almaden Testdatengenerator vorgestellt und die Gründe dargelegt, warum dieser Generator für E-Commerce Lösungen nicht geeignet ist.

In den folgenden Kapiteln werden die theoretischen Grundlagen des in dieser Arbeit entwickelten Testdatengenerators beschrieben, wobei immer wieder auf die Anwendung dieser theoretischen Konzepte im entwickelten Testdatengenerator hingewiesen wird. In Kapitel 3 wird die Grundlage des Generators, die Repeat Buying Theorie von Ehrenberg dargelegt. In Kapitel 4 wird näher auf die Berechnung der Transaktionen im Testdatengenerator eingegangen. In Kapitel 5 werden die

Algorithmen zur Ermittlung von Zufallszahlen grundsätzlich erläutert und jene, die für den Testdatengenerator verwendet wurden näher beschrieben. Ebenso werden die beiden für die Ermittlung der Parameter relevanten Verteilungen - die Exponentialverteilung und die Gammaverteilung diskutiert.

Kapitel 6 – 8 beschäftigen sich mit dem erzeugten Testdatengenerator und umfassen die Programmdokumentation, die Auswertung einiger erzeugter Testdaten sowie den Programmcode.

1 Die Grundlagen des Data Mining

1.1 Knowledge Discovery in Databases

Mit zunehmend ausgereifteren Technologien in der Informatik und damit auch immer mächtigeren Werkzeugen im Bereich Datenbanksysteme fallen immer mehr Daten verschiedenster Qualität und größerer Quantität an. Diese Datenmengen zu analysieren und entsprechende Informationen zu gewinnen ist die primäre Aufgabe des Data Mining. Es ist jedoch nicht als isolierter Arbeitsschritt zu sehen. Vielmehr stellt es ein Teilstück in einem umfassenden Prozess, dem Knowledge Discovery in Databases (KDD) dar. In [FPSSU96] ist folgende Definition für KDD zu finden:

Knowledge Discovery in Databases is the non-trivial process of identifying valid, novel, potentially usefull and ultimately understandable patterns in data.

KDD ist also der Vorgang, in Datenmengen Muster zu finden. Wobei diese Muster laut der Definition, gültig, neu, nützlich und verständlich sein sollen.

Die Daten, die analysiert werden sollen liegen meist in verschiedenen Formaten und aus verschiedenen Quellen vor. Alle Datenquellen werden in Data Warehouses zusammengefasst und eine Vorauswahl der Daten getroffen. Die gewonnen Daten werden so transformiert, dass sie für den nachstehenden Data Mining Prozess als Ausgangsdaten zur Verfügung stehen.

Die ersten Anwendungsgebiete für Data Mining waren Transaktionsdaten, die aus Supermarktkassen gewonnen werden. Ein Datensatz dieser Transaktionsdaten besteht zumindest aus einem Zeitstempel und einer Liste von Artikel die zusammen gekauft wurden. Also ein Einkauf in einem Supermarkt wird durch den Kassabon dargestellt. Dieser enthält die Liste der gekauften Artikel sowie das Datum und die Uhrzeit des Einkaufes.

Durch das Aufkommen des Internets versuchte man auch das Surfverhalten der User im WWW mittels Data Mining zu erforschen. Ein wichtiges Problem dabei ist festzulegen, was als eine Transaktion zu gelten hat. Von einem Webserver werden Aufzeichungn geführt (Log-Daten) welche Seiten von welcher IP-Adresse aus angesurft wurden. Allerdings reicht es nicht aus, anhand der unterschiedlichen

Adressen Transaktionen abzugrenzen. Durch Einwahl bei einem Internetprovider kann eine IP-Adresse mehreren Nutzern zugewiesen werden; jedoch nicht gleichzeitig. Daher wird meistens eine Transaktion durch idente IP-Nummer und eine gewisse Zeitspanne definiert. Z.B. alle Seiten die von der IP Nummer 192.168.0.1 im Zeitraum von 5 Minuten angesurft wurden, werden zu einer Transaktion zusammengefasst. Nach dieser Methode kann es jedoch vorkommen, dass ein User eine Seite 6 Minuten lang liest und dann die folgende Seite ansurft. Die beiden Seitenaufrufe würden auf zwei Transaktionen aufgeteilt werden, was jedoch nicht richtig ist, da es ein User war.

Die Regel, wie aus den Log-Daten des Webservers Transaktionsdaten gebildet werden, hängt von der angebotenen Information und der eingesetzten Technologie ab. Z.B. kann mittels Cookies das Surfverhalten eines Users besser ausgewertet werden. Dies muss jedoch schon bei der Erstellung der Website berücksichtigt werden. Auf dieses und weitere Probleme sei hier nur hingewiesen, in dieser Arbeit wird nicht näher darauf eingegangen. Eine Aufstellung der Probleme findet man z.B. in [Hahs02].

1.2 Data Mining Verfahren

Für die oben erwähnten Transaktionsdaten, die aus Webserver-Logs oder von Scannerkassen gewonnen werden, können nun mittels diverser Data Mining-Verfahren Zusammenhänge von Produkten in den Transaktionen gefunden werden. D.h. es werden Produkte gesucht, die oftmals gleichzeitig in einer Transaktion erworben werden.

1.2.1 Grundlagen der Assoziationsregeln

Eines der am weitest verbreiteten Verfahren ist die Suche nach Assoziationsregeln (engl. mining association rules). Aufgrund der gefundenen Regeln möchte man vom Vorhandensein eines Produktes in einer Transaktion auf das Vorhandensein eines anderen Produktes in der gleichen Transaktion schließen können, wobei es natürlich vor allem interessant ist, versteckte, nicht offensichtliche Regeln zu entdecken.

Die entdeckten Assoziationsregeln können zum Beispiel bei der Entwicklung von Marketingstrategien hilfreich sein. Im Falle einer Website können damit dem Nutzer

der Website weitere Seiten vorgeschlagen werden, die zu dem Themengebiet der aktuellen Seite passen.

Um das Problem des Suchens nach Assoziationsregeln beschreiben zu können, wurde in [FPSSU96] ein mathematisches Modell vorgeschlagen. In diesem Modell wird eine Regel durch eine Schlussfolgerung der Form X => Y dargestellt, wobei X und Y jeweils für bestimmte Artikel (engl.: items) stehen. Das Modell enthält zwei Faktoren, die für die Bestimmung von Assoziationsregeln wichtig sind:

1. Die Konfidenz c (engl.: confidence) beschreibt die Stärke der Abhängigkeit. Ein Wert von 50% bedeutet zum Beispiel, dass in jeder zweiten der Transaktionen die X enthalten auch Y vorkommen.

2. Die Unterstützung s (engl.: support) zeigt die Häufigkeit des Vorkommens einer Artikelkombination in den untersuchten Transaktionen. D.h. wie oft ist eine Assoziationsregel in einer Menge von Transaktionen erfüllt?

Normalerweise schenkt man nur den Regeln Beachtung, die über ein ausreichend hohen Konfidenzwert und eine starke Unterstützung verfügen. Diese Regeln werden auch als „starke Regeln“ (engl.: strong rules) bezeichnet.

1.2.2 Generischer Algorithmus

Aufgrund der großen Datenmengen kommt dem Algorithmus zur Findung der Regeln große Bedeutung zu. Die Art des Algorithmus bestimmt wesentlich die notwendigen Ressourcen für die Auswertung der Datenbestände. Großes Augenmerk muss auf die Ausführungsdauer gelegt werden. Jedoch sind alle Algorithmen nach folgendem Prinzip aufgebaut [Baum98]:

1. Finde alle „häufigen Artikelmengen“ (oftmals engl.: frequent oder large itemsets) in der Datenbasis D

2. Generiere aus diesen Mengen die Regel, welche die vorgegebene Konfidenz c (confidence) und die notwendige Unterstützung s (support) erfüllen.

3. Filtere die Regeln nach Interesse des Benutzers

Ad 1: Das Finden der häufigen Artikelkombinationen bildet den Kern aller Verfahren. Hierbei treten die größten Unterschiede zwischen Algorithmen auf. Wobei der verwendete Algorithmus auch von der Art der gesuchten Assoziationsregel abhängt. (siehe 1.2.3 Arten von Assoziationsregeln)

Ad 2: Nachdem die Artikelmengen gefunden wurden, können die Regeln gebildet werden.

Ad 3: Meistens wird eine große Anzahl an Regeln gefunden, wobei jedoch nicht alle für den Benutzer interessant sind. Daher ist es angebracht, die Regeln zu filtern. Teilweise können Regeln verworfen werden, da sie von anderen Regeln bereits mit abgedeckt werden.

1.2.3 Arten von Assoziationsregeln

1.2.3.1 Boolsche Assoziationsregeln

Es gibt verschiedene Verfahren zur Bildung der Regeln. Regeln, die über boolsche Attributwerte gebildet werden, nennt man boolsche Assoziationsregeln. Hierbei wird nur festgestellt, ob ein Artikel in der Transaktion enthalten ist oder nicht.

Zu den bekanntesten Algorithmen dieser Art zählt das AIS-Verfahren das 1993 in [AIS93] vorgestellt wurde und als einer der ersten Algorithmen betrachtet wird.

SetM ist ein weiterer bekannter Vertreter dieser Art. Der Ansatz in diesem Algorithmus ist, die Regeln mit der Datenbankabfragesprache SQL zu koppeln. Der komplette Algorithmus lässt sich mit SQL Befehlen beschreiben. (siehe [HS93]).

Ein weiterer bekannter Algorithmus ist der APRIORI-Algorithmus [AGRSRI94]. Von diesem Algorithmus gibt es Erweiterungen, die versuchen einige Schwachstellen, insbesondere den Hauptspeicherbedarf zu verbessern, arbeiten jedoch nach dem gleichen Grundprinzip.

1.2.3.2 Generalisierte Assoziationsregeln

Oftmals ist es interessant, zwischen einzelnen Artikeln Taxanomien zu definieren. Eine Taxanomie ist eine ist-ein-Beziehung zwischen zwei Artikeln. Z.B.: Eine

Haltbarmilch ist eine Milch; Ein Brot ist ein Lebensmittel. In vielen Fällen werden so neue, noch nicht entdeckte Regeln gefunden.

Zur Veranschaulichung soll das Beispiel in Abbildung 1 dienen. Es ist schwierig, zwischen dem Kauf von Baguette und dem Kauf von Rotwein eine starke Unterstützung s festzustellen. Demgegenüber ist es gut möglich, dass viele der Kunden die Brot einkaufen auch Wein kaufen. Außerdem könnte es auch interessant sein, dass Kunden, die Baguette kaufen, mit hoher Wahrscheinlichkeit Rotwein einkaufen.

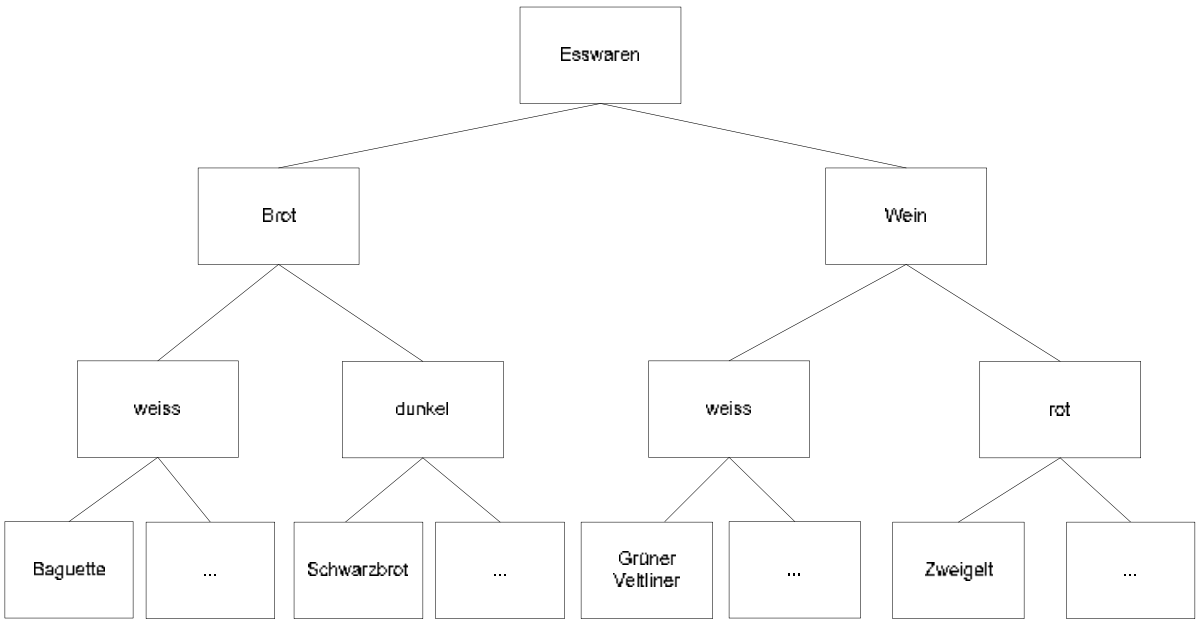


Abb: 1 Beispiel für Taxanomien zwischen zwei Artikel

Solche Regeln findet man mit Hilfe der generalisierten Assoziationsregeln. Hierbei ist ein Mehraufwand nötig, um die Strukturen zu erstellen. Auch sind die Algorithmen komplexer, da es nun nicht nur notwendig ist, festzustellen ob ein Artikel in der Transaktion enthalten ist. Es muss zusätzlich noch geprüft werden, in welche Hierarchiestufe der Artikel fällt.

Ein Algorithmus zum Finden der generalisierten Assoziationsregeln ist das BASIC-Verfahren [SRIAGR95]. Das ist ein leicht veränderter APRIORI- Algorithmus. Auch in diesem Bereich gibt es Weiterentwicklungen, wobei immer die Ausführungszeit und der Ressourcenbedarf optimiert werden.

Bei der Entwicklung der Algorithmen ist es notwendig, Zugriff auf einen großen Datenbestand zu haben. Daten die aus echten Systemen genommen werden, können z.B. durch saisonale Schwankungen oder durch Marketingaktionen des Handelsbetriebes verfälscht sein und eignen sich daher nicht, um die Effizienz eines Algorithmus festzustellen. Daher wird man im Zuge der Entwicklung versuchen Datenbestände zu generieren, die gewissen Regeln folgen und diese künstlichen Daten für die Testläufe des Algorithmus heranziehen.

2 IBM's Almaden - Testdatengenerator

2.1 Almaden

IBM betreibt weltweit für Forschungsaufgaben acht eigene Forschungseinrichtungen. Eine dieser Einrichtung ist das Almaden Research Center in der Nähe von San Francisco. Dieses Research Center ist innerhalb des IBM Konzerns für die Bereiche Database Research und Data Storage zuständig. In den siebziger Jahren haben Ingeneure von Almaden das Konzept der relationalen Datenbank erfunden. [Alma02]

Ein bereits abgeschlossenes Projekt ist „Quest". Die Quest-Arbeitsgruppe beschäftigte sich unter anderem auch mit „Data Mining" [Quest02]. Im Zuge der Arbeit der Gruppe wurde auch ein Testdatengenerator entwickelt dessen Source-Code unter http://www.almaden.ibm.com/cs/quest/syndata.html heruntergeladen werden kann. Der Code wurde in Standard C++ geschrieben und sollte daher unter allen Betriebssystemen kompilierbar sein, wobei jedoch das Make-File nur für AIX vorliegt.

Eine Dokumentation über die Programmlogik ist auf der Website nicht verfügbar. Jedoch wird in [AGRSRI94] die Erstellung der Daten beschrieben. In weiterer Folge wird hier kurz auf die Arbeitsweise dieses Testdatengenerators eingegangen.

2.2 Arbeitsweise des Generators

Der Generator erzeugt Datensätze, die den Transaktionen von Konsumenten in einer Einzelhandelsumgebung entsprechen. In dem Modell des Generators kaufen die Personen Produktfolgen (engl. sequence) von Produktgruppen. So werden z.B Leintücher, Kopfkissen und Bettdecken gekauft. Es kann jedoch auch vorkommen, dass nur Teile einer Produktfolge gekauft werden. So werden auch nur Leintücher alleine gekauft.

Anderseits kann eine Person auch Teile aus mehreren Folgen zusammen kaufen. So kann es vorkommen, dass Leintücher und Kopfkissen zusammen mit einer Jacke und Hose gekauft werden, wobei Jacke und Hose aus einer anderen Folge stammen.

Die Größen dieser Kunden-Produktfolgen sind normalerweise rund um einen einstellbaren Wert gedrängt. Zusätzlich haben nur wenige Kunden viele Transaktionen zugeordnet. Weiters sind die Transaktionsgrößen in diesem Modell ebenfalls um einen Wert gedrängt und auch hier haben nur wenige Transaktionen viele Produkte.

Der Generator arbeitet mit folgenden Parametern die beim Programmstart eingestellt werden können:

$\|D\|$	Anzahl der Kunden
$\|C\|$	Durchschnittliche Anzahl der Transaktionen je Kunde
$\|T\|$	Durchschnittliche Anzahl der Produkte je Transaktion
$\|S\|$	Durchschnittliche Länge der Produktfolgen
N_s	Anzahl der maximalen Produktfolgen

Zunächst werden für den zu berechnenden Kunden die Anzahl der Transaktion und die durchschnittliche Größe einer Transaktion bestimmt. Die Anzahl der Transaktionen wird aus einer Poisson-Verteilung mit einem μ gleich |C| gezogen. Die durchschnittliche Größe einer Transaktion wird ebenfalls aus einer Poisson-Verteilung mit einem μ gleich |T| gezogen.

Im nächsten Schritt werden der Transaktion des Kunden einige Produktfolgen von Item-Sets zugewiesen. Wenn eine Folge nicht mehr in die Transaktion passt wird in 50 % der Fälle diese Produktfolge der Transaktion trotzdem zugewiesen; in den anderen 50 % wird diese Produktfolge für die nächste Transaktion des Kunden aufgehoben.

Die Produktfolgen werden aus einer Tabelle ? genommen. Die Anzahl der Produktfolgen in ? ist gleich dem Parameter N_s. Für die Bildung einer Produktfolge in ? wird zunächst die Anzahl der Produkte in dieser Folge bestimmt indem eine Zufallsvariable aus einer Poission-Verteilung mit einem Mean μ gleich |S| gezogen wird.

Die Produktgruppen in der ersten Produktfolge werden zufällig aus einer Liste der Produktgruppen gezogen. Die Liste der Produtkgruppen (large Itemsets) ist ähnlich der Liste der Produktfolgen. Da die aufeinander folgende Produktfolgen meistens Produkte gemeinsam haben, werden Teile von Produktgruppen, die in einer Folge verwendet wurden ebenfalls in der nachfolgenden Produktfolge nochmals verwendet. Es wird eine exponentiell verteilte Zufallsvariable mit einem Mean gleich dem „correlation level“ verwendet um den weiterverwendeten Anteil jeder Produktgruppe zu bestimmen. Die restlichen Produktgruppen für die Folge werden zufällig ausgewählt.

Jede Folge in ? wird gewichtet, sodass das Gewicht die Wahrscheinlichkeit ausdrückt, mit der diese Folge gewählt wird. Die Gewichte werden aus einer Exponential-Verteilung gezogen. Alle Gewichte werden normalisiert, sodass die Summe alle Gewichte der Folgen in ? einen Wert von 1 ergibt.

Nachdem nicht alle Produkte einer Produktfolge immer zusammen gekauft werden, wird jeder Folge in ? ein „corruption level“ c zugewiesen. Wenn eine Folge einer Transaktion eines Kunden zugewiesen wird, werden solange Produkte aus der Folge verworfen, bis eine gleich verteilte Zufallszahl im Bereich von 0 bis 1 kleiner ist, als der „corruption level“ c.

Die Ausgabe der erzeugten Testdaten kann auf zwei verschiedene Arten erfolgen werden. Bei der Binär-Ausgabe werden die Werte <CustID, TransID, NumItems, List-Of-Items> als 4-Byte Zahlen ausgegeben. D.h. man erhält pro Transaktion bereits eine aufbereitete Produktliste.

Bei der Text-Ausgaben werden die Werte in der Form <CustID, TransID, Item> pro Zeile ausgegeben wobei jeder Wert aus 10 Zeichen (Byte) besteht und somit für jede Zeile inklusive der Trennzeichen 33 Byte notwendig sind und man keine Zusammenstellung der Produkte pro Transaktion erhält.

Weiters wird ein „Pattern“ File erstellt. Dieses enthält eine Beschreibung der Daten, die verwendeten Ausgangsparameter sowie die gebildeten Itemsets und die Gewichtung dieser Itemsets.

Dieser Testdatengenerator arbeitet mit einem Modell welches für reale Güter entwickelt worden ist. Es werden Produktgruppen gebildet und diese gemeinsam gekauft. In Onlineshops und auf Websites herrschen jedoch Transaktionen vor, die nur aus einem Produkt (=Seitenzugriff) bestehen. Daher sind Daten, die von diesem Generator produziert werden nur bedingt für die Analyse von Algorithmen geeignet, die die Zugriffe auf Websites auswerten sollen. [ZhKoMa01]

3 Ehrenbergs Repeat Buying Theorie als Basis zur Entwicklung des Testgenerators

Chatfield, Ehrenberg und Goodhart untersuchten schon 1970 Verkaufszahlen, um auf Regelmäßigkeiten im Kaufverhalten der Kunden zu stoßen. Aus den gewonnen Erkenntnissen entwickelten sie das Negativ Binominal Distribution (NBD) Modell. Auf diesem basiert die grundlegende Idee des Testdatengenerators der in dieser Arbeit erstellt wird. Daher wird im Folgenden dieses Model näher vorgestellt.

Das NBD Modell ist ein Modell, welches auf alle häufig gekauften Konsumgüter anwendbar ist. Es bietet ein Muster, in welchen Abständen Kunden Güter kaufen. Dabei ist es nicht relevant um welche Güter oder Marken es sich handelt.

Das Kaufverhalten wird von vielen Faktoren, wie z.B. Werbung, Preis, Verfügbarkeit und Verpackungsgröße [Ehren88, S.3] beeinflusst. Der Käufer entscheidet, welches Produkt, zu welchem Preis wann und wo gekauft wird. Bei Niedrigpreis-Produkten entwickeln die Käufer eine gewisse Gewohnheit um nicht bei jedem Kauf alle Entscheidungen nochmals zu treffen. Somit kommt es zu einer gewissen Markentreue. Es wird meistens die Marke gekauft, die das letzte Mal gekauft wurde. Ehrenberg konzentriert sich bei seinen Auswertungen auf die Kaufzeitpunkte und lässt die anderen Faktoren wie zum Beispiel Packungsgröße und Menge außer Acht.

Bei der Analyse des Kaufverhaltens eines Haushaltes treten folgende Fragen auf

1) Warum wurde etwas gekauft?

2) Mit welcher Regelmäßigkeit wurde etwas gekauft?

Ehrenbergs Repeat Buying Theorie beschreibt die Regelmäßigkeit und gibt keine Antwort auf das Warum.

Ehrenberg verwendet für seine Studien als Ausgangsdaten so genannte „consumer panels“. Hierbei führt eine ausgewählte Gruppe über alle Einkäufe Buch und berichtet in regelmäßigen Abständen an eine zentrale Stelle. Von dieser Stelle können die gesammelten Daten erworben werden. Der Vorteil an „consumer panels“ ist, dass es so möglich ist, über einen längeren Zeitraum die Einkäufe eines

Konsumenten zu verfolgen und so das Kaufverhalten festzustellen. Andere Auswertung, wie z.B. Scannerkassen liefern nur Daten die auf Produkte bezogen sind, und aus diesen Daten kann nicht auf den einzelnen Konsumenten geschlossen werden.

Ein Grund für die zunehmende Überschwemmung mit Kundenbindungsprogrammen und Kundenkartenprogrammen liegt in der Notwendigkeit festzustellen was ein Kunde kauft. Um nicht nur auf einen Kaufvorgang beschränkt zu sein findet die Verknüpfung aller Kaufvorgänge eines Kunden oder eines Haushaltes über die Kundenprogramme statt. Somit erhält man von den Scannerkassen nicht nur was pro Transaktion gekauft wurde, sondern kann auch aus den Datenmengen herausfinden wie sich das Kaufverhalten eines Kunden über die Zeit verändert

Ehrenberg bringt in [Ehren88, S 7] folgendes Beispiel für die Verwendung der Daten:

Die Tabelle zeigt die Einkäufe für 4 Haushalte über 12 Wochen. Dabei wurden jedoch nur die Produkte A und B betrachtet. Consumer Panels enthalten weitere Informationen wie Preis, Geschäft und gekaufte Menge. Für die Betrachtungen wurde jedoch nur die Tatsache ob etwas gekauft wurde herangezogen.

Woche	1	2	3	4	5	6	7	8	9	10	11	12
Haushalt 1	-	-	-	-	A	-	A	A	A	B	-	A
Haushalt 2	A	-	A	A	-	-	-	-	-	-	-	-
Haushalt 3	-	B	-	-	-	-	-	B	A	-	-	-
Haushalt 4	-	-	-	-	-	-	-	-	-	-	-	-

Als Nächstes wird ein größerer Betrachtungszeitraum gewählt. Es werden jeweils 4 Wochen zu einem Monat zusammengefasst. Die aggregierte Tabelle hat folgenden Inhalt:

Monat	1	2	3	Summe

Haushalt 1	-	3A	2A,1B	5A,1B
Haushalt 2	3A	-	-	3A
Haushalt 3	1B	1B	1A	1A,2B
Haushalt 4	-	-	-	
Summe	3A,1B	3A,1B	3A,1B	

In dieser Ansicht erkennt man große Differenzen im Einkaufsverhalten der einzelnen Haushalte. Summiert man jedoch die Einkäufe für jedes Monat, so erkennt man, dass die Nachfrage nach A und B jedes Monat gleich ist und man kann daraus schließen, dass im vierten Monat ebenfalls 3 A und 1 B gekauft werden.

Ehrenberg stellte fest, dass die früheren Ansätze zu kompliziert waren und zu viele Kriterien in die Auswertungen mit einbezogen. Er versucht daher eine Vereinfachung vorzunehmen um ein Modell zu erhalten, das auf die meisten Produkte zutrifft.

Das erste Kriterium ist die Einheit die betrachtet wird. Mögliche Einheiten wären die Menge oder der Wert der von einem Produkt gekauft wird. Im ersten Fall muss man die unterschiedlichen Packungsgrößen betrachten. Toast wird in Familienpackungen und in „Single“-Packungen verkauft. Der Wert beinhaltet auch den Preis zu dem gekauft wird und dieser kann sehr stark variieren. Ehrenberg betrachtet als Einheit nur die „purchase occasion“. D.h. nur die Tatsache dass ein Produkt gekauft wird findet Verwendung in seiner Theorie. Die Menge und Packungsgröße wird außer acht gelassen.

Käufer können ihre Bedürfnisse auch mit gleichen Produkten einer anderen Marke befriedigen. D.h. ein Käufer kann die Milch einmal vom Hersteller X und dann vom Hersteller Y kaufen. Es gab Ansätze diese Produktklassen auszuwerten. Ehrenberg beschränkt seine Auswertungen darauf, nur Produkte und keine Produktklassen zu betrachten.

Der dritte Auswertungsaspekt ist der Zeitraum. Ehrenberg wertet die Daten in gleichen Zeiträumen aus. (z.B: Wochen, Monate oder Jahre). Alternativ könnte man die Sequenz von Einkäufen auswerten. Ein Problem dabei ist, das einzelne Käufe

die Sequenz durcheinander bringen können. Durch die Auswertung der Zeiträume umgeht man dieses Problem. Weiters ermöglicht diese Vorgangsweise, dass das Kaufverhalten durch nur zwei Parameter bestimmt werden kann: [Ehren88 S.11] “Penetration“ (b): Anteil der Personen die zumindest einmal das Produkt in einer Periode kaufen und „Purchase Frequency“ (w): die durchschnittliche Anzahl wie oft diese Käufer das Produkt in der Periode kaufen.

Mit diesen Vereinfachungen hat Ehrenberg die „Consumer Panels“ untersucht und dabei festgestellt, dass die Werte, die durch das NBD-Modell errechnet wurden zum Großteil mit der Realität übereinstimmten. [Ehren88 S.57] Eine Vorraussetzung ist jedoch, dass es zu keinen Schwankungen aufgrund von Saisonen oder kurzfristigen Marketingaktion kommt.

Eine weitere Vereinfachung des NBD-Modells wird durch die Verwendung einer anderen Verteilung, der Logarithmic Series Distribution (LSD), erzielt. Diese Vereinfachung kann unter einer bestimmten Voraussetzung angewandt werden: Der Anteil der Käufer in der Menge der beobachteten Käufer muss unter 20 % liegen und der Parameter w, die „Purchase Frequency“ muss über 1,5 liegen. Der entscheidende Punkt ist, dass das LSD-Modell nur die „Buyer“ betrachtet und die „Non-Buyer“ außer Acht lässt, was zulässig ist, solange der Anteil an „Non-Buyer“ hoch genug ist.

Wie gut die beiden Modelle mit beobachteten Daten übereinstimmen zeigt folgende Tabelle die aus [Ehren88] entnommen wurde:

Anzahl der Käufe	Nach LSD-Modell	Beobachtet	Nach NBD-Modell
0	(1612)	1612	(1612)
1	165	164	157
2	72	71	74
3	42	47	44
4	27	28	29
5	19	17	20
6	14	12	15
7	10	12	11
8	8	3	8
9	6	7	6
10	4	6	5
11	3	3	4
12	3	3	3
13	2	5	2
14	2	0	2
15	2	0	1
16	1	0	1
17	1	2	1
18	1	0	1
19	1	0	1
20	1	1	1
21+	4	5	2

Der Anteil an „Non-Buyers“ $p_0 = .806$

Anteil an „Buyers“: $b = 1-p_0 = 0.194$

Durchschnittliche Anzahl an Käufen pro Haushalt: $m = 0.636$

Durchschnittliche Anzahl an Käufen pro Käufer: $w = 3.3$

Um die Anzahl der Käufer zu bestimmen die ein Produkt in einem Zeitraum r-mal gekauft haben verwendet man folgende Formel:

$$p'_r = p_r / (1-p_0)$$

Das Mean der Verteilung (m) wird durch w ausgedrückt wobei gilt:

$$w= m/(1-p_0)$$

Hierbei steht p'_r für die Anzahl der Käufer die das Produkt in einer Periode r mal gekauft haben und p_r steht für die Anzahl im gesamten Betrachtungszeitraum.

Im LSD-Modell wird p'r nach folgender Formel bestimmt:

$$p'_r = -1 / \ln(1-q) * q^r / r$$

Der Parameter q wird in folgender Formel ausgedrückt:

$$w = -q / ((1-q) \ln(1-q))$$

Diese Formel kann nicht direkt für q aufgelöst werden. Um aus einem gegebenen w ein q zu berechnen, kann man Tabellen in [Ehren88] verwenden oder folgende Formel für eine Näherungsberechnung verwenden:

$$q \tilde{} \ (w-1.4) / (w-1.15)$$

Somit kann mittels des LSD-Modells die Berechnung mit nur einem Parameter, der „Purchase frequency" (w) ermittelt werden. Aufgrund der genauen Ergebnisse des LSD-Modells kann zur Berechnung durchaus das vereinfachte Modell herangezogen werden solange die folgenden Voraussetzungen erfüllt sind.

Der Anteil der „Non-Buyer" ist ausreichend groß. Die Käufe eines Kunden erfolgen zufällig und folgen somit einem Poisson-Prozeß und die Mean-Werte der Poisson-Prozesse aller Kunden sind ?-Verteilt.

In dem Fall des Testdatengenerators werden Zugriffe auf Webseiten simuliert. Die Vorraussetzung des LSD-Modells treffen für Websites zu. Somit wird im Testdatengenerator mit dem einfacheren LSD-Modell gearbeitet.

4 Berechnung der Transaktionen

Aufbauend auf dem LSD-Modell der Repeat Buying Theorie von Ehrenberg, wurde ein Modell entwickelt, wie die Testdaten dieses Generators erzeugt werden. Die Vorgangsweise wird in diesem Kapitel näher beschrieben.

4.1 Ausgangsdaten

4.1.1 Produkte

Zunächst wird beim Start des Generators eine Menge an Produkten gebildet. Aus dieser Menge kann jeder Kunde seinen Einkauf zusammenstellen. Die Anzahl der Produkte ist beim Programmstart als Parameter festzulegen. Die Produkte werden innerhalb des Programms und bei der Ausgabe in die Datei anhand einer fortlaufenden, ganzen Zahl identifiziert.

Da nicht jedes Produkt gleich oft gekauft werden soll, wird bei der Erzeugung der Produkte ein Parameter „Mean“ bestimmt. Dieser wird mittels gleich verteilter Zufallszahl aus einem bestimmbaren Werte-Band gefunden. Gibt man beim Aufruf einen Bereich von 0-4 an, so bedeutet dies, dass jedes Produkt einen Wert Mean zwischen 0 und 4 hat. Diese Werte sind über alle Produkte gleich verteilt und es kommt somit nicht zu einzelnen Spitzen.

4.1.2 Kunden

Die Anzahl der Kunden wird ebenfalls beim Programmstart angegeben. Die nachfolgenden Schritte zur Bestimmung der Transaktionen eines Kunden werden für jeden angegebenen Kunden in einer Schleife wiederholt.

4.1.3 Zeitraum

Als Zeitraum, der den Kunden für den Einkauf zur Verfügung steht wurde ein Intervall von 0 bis 1 gewählt. Somit können die erzeugten Daten jederzeit auf ein beliebiges anderes Intervall umgerechnet werden. Zur Simulation von Einkäufen in Geschäften müsste darauf geachtet werden, dass Sonntags die Geschäfte geschlossen sind und es dürften keine Transaktionen an Sonntagen erzeugt werden. Die Daten stellen jedoch primär Seitenzugriffe auf Websites dar. Daher konnte die

Thematik Sonntage und Feiertag bei der Erzeugung der Daten vernachlässigt werden. Nachtzeiten werden ebenfalls nicht berücksichtigt.

4.2 Kundenverhalten

Bei der Erzeugung der Transaktionsdatensätze wird von folgender Annahme ausgegangen: Ein Kunde entwickelt diverse Kaufbedürfnisse für einzelne Produkte. Diese Bedürfnisse werden über einen Zeitraum gesammelt. Alle aufgestauten Bedürfnisse in einem kurzen Zeitraum werden auf einmal gestillt. D.h. der Kunde geht einkaufen und besorgt die einzelnen Produkte zugleich. Dieser Zeitraum, wie lange ein Kunde die Kaufbedürfnisse aufstaut, kann beim Programmaufruf eingestellt werden, ist jedoch für alle Kunden gleich.

In Kapitel 6.3.4 Ansätze für Programmerweiterungen wird darauf eingegangen wie ein anderes Kaufverhalten der Kunden implementiert werden kann.

4.2.1 Kaufzeitpunkte

Im ersten Schritt werden nun für jedes Produkt in der Produktmenge von dem Kunden seine Kaufwünsche definiert. D.h. es wird bestimmt, wann der Kunde Bedarf nach dem Produkt hat. Da dies für jedes Produkt unterschiedlich ist, wird der Mean-Parameter des Produktes genommen und als Parameter für eine Gammaverteilung herangezogen. Danach wird aus dieser Gammaverteilung ein Wert gezogen. Dieser Wert gibt an, wie oft in dem Zeitintervall der Kunde das Produkt nachfragen wird und beschreibt somit das Kaufverhalten des Kunden bezüglich eines Produktes.

Es werden in weiterer Folge die einzelnen Zeitpunkte auf der Zeitachse bestimmt zu denen der Kunde einen Bedarf für das Produkt entwickelt. Hierzu wird ein Poisson-Prozess angewandt, dessen Parameter das bestimmte Kaufverhalten ist. Die Abstände zwischen den einzelnen Zeitpunkten sind exponential verteilt. Wobei das Kaufverhalten den Parameter der Exponentialverteilung darstellt.

Nach dem Durchlauf für ein Produkt sieht die Zeitachse eines Kunden folgendermaßen aus:

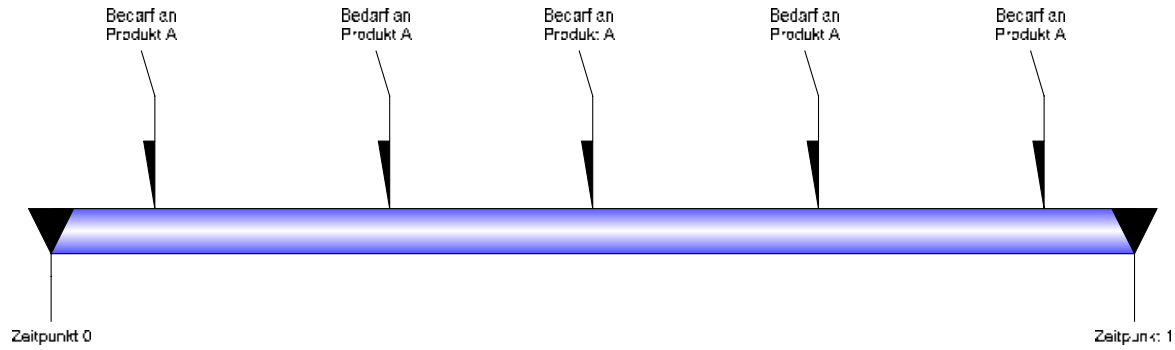


Abb: 2 Einkauf eines Produktes

Dieser Vorgang wird für jedes Produkt wiederholt. Nach allen Produkten weist die Zeitachse eines Kunden folgendes Bild auf:

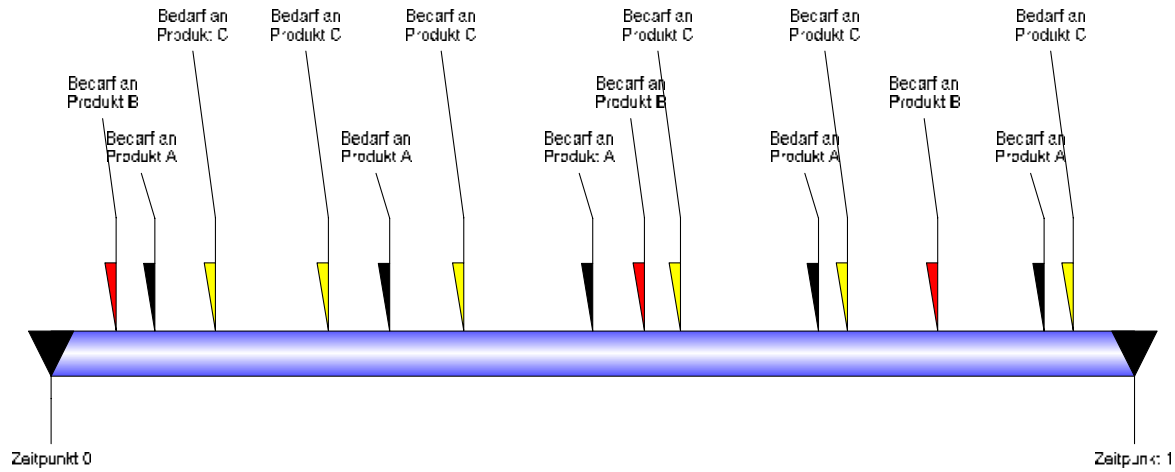


Abb: 3 Zeitachse nach allen Produkten

4.2.2 Bildung der Transaktionen

Sobald für einen Kunden alle Kaufzeitpunkte bestimmt sind, werden aus der Zeitachse die Transaktionen bestimmt. Hierbei wird vom Zeitpunkt 0 ausgehend das Zeitintervall für eine Transaktion betrachtet. Alle Kaufwünsche die in diesen Zeitraum fallen, werden in einer Transaktion zusammengefasst. Der letzte Zeitpunkt in dieser Transaktion wird als Zeitpunkt der Transaktion verwendet.

Das Betrachtungsintervall für die nächste Transaktion beginnt am Ende der zuletzt gebildeten Transaktion.

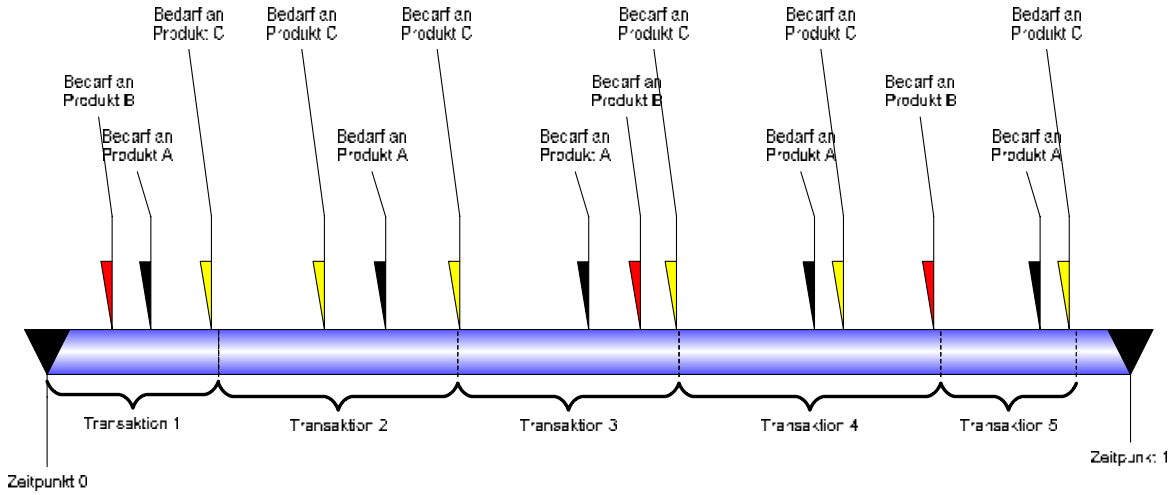


Abb: 4 gebildete Transaktionen

Nach der Bildung der Transaktionen für die oben angegebenen Zeitachsen hat dieser Kunde folgende Transaktionsliste in Tabelle 1:

Transaktionsnummer	Zeitpunkt	Produktliste
1	0,1	A,B,C
2	0,3	A,C,C
3	0,5	A,B,C
4	0,75	A,B,C
5	0,8	A,C

Tabelle 1: Beispiel der Transaktionsliste des Kunden

5 Algorithmen zur Findung von Zufallszahlen

Die Qualität der erzeugten Daten hängt zu einem großen Teil von dem verwendeten Zufallszahlengenerator ab. Daher ist die Auswahl des verwendeten Zufallsgenerators wichtig. Zunächst muss bestimmt werden, welche Zufallsverteilung für den Testdatengenerator verwendet wird.

5.1 Die benötigten Zufallszahlenverteilungen

5.1.1 Exponentialverteilung

Der Zeitpunkt der Einkäufe eines Kunden in einem Zeitraum folgt dem Poisson-Prozess das bedeutet, dass innerhalb des Zeitraumes die Zeitabstände zwischen den Einkäufen exponentialverteilt sind.

Für einen Poisson-Prozess müssen einige Voraussetzungen [RoCle96] erfüllt sein:

1.) Es gibt viele mögliche Punkte auf der Zeitachse zu denen die Ereignisse eintreten können. Die Zeitachse kann jedoch eine beliebige Einheit haben. Daher wurde in dem Generator eine Zeitachse von 0 bis 1 gewählt. Je nachdem womit 1 gleichgesetzt wird, kann man die gewonnen Daten in einem Zeitraum von einem Jahr, einem Monat oder einer Woche betrachten.

2.) An jedem Punkt der Zeitachse ist die Wahrscheinlichkeit, dass das Ereignis eintritt klein. Das bedeutet, dass die Anzahl der Ereignisse in dem gesamten Zeitraum nicht sehr hoch ist. Umgelegt auf den Kundenstrom in einem Geschäft würde dies bedeuten, dass die Kunden über den Zeitraum verteilt in das Geschäft kommen und nicht zu einer bestimmten Zeit, wie es bei einem Bäcker die Morgenzeit sein wird.

3.) Die Ereignisse sind untereinander unabhängig. Also zu jeder Zeit ist die Wahrscheinlichkeit, dass ein Ereignis eintritt gleich hoch.

4.) Die durchschnittliche Anzahl von Ereignissen pro Zeiteinheit ist gleich hoch, egal wie lange der Prozess bereits läuft.

Die Poisson Verteilung ist eine diskrete Verteilung. Als Ergebnis erhält man eine Zahl. Z.B. in einer Stunde kommen 10 Personen in ein Geschäft. Da die Zeit

zwischen den einzelnen Ereignissen bei einem Poisson-Prozess exponential verteilt ist wird zur Erzeugung der Testdaten zunächst eine exponential verteilte Zufallszahl benötigt.

5.1.2 Gammaverteilung

Nicht jeder Kunde kauft ein bestimmtes Produkt A gleich oft. Einige Kunden kaufen dieses Produkt gar nicht, andere Kunden kaufen das Produkt sehr oft. Um diese Schwankungen simulieren zu können, wurde ein Wert „Kaufverhalten“ für jede Kombination (Kunde, Produkt) eingeführt. Diese Werte des „Kaufverhaltens“ sind für alle Kunden über ein Produkt Gamma-verteilt. Mit dieser Verteilung werden auch Werte generiert die bedeuten, dass ein Kunde ein Produkt niemals kauft. Diese sind die Non-Buyer eines Produktes. Genauso werden wenige Kunden ein Produkt sehr oft kaufen.

5.2 Zufallszahlengeneratoren

Um ausreichende Transaktionsdaten für Simulationen von zufallszahlenbeeinflußten Systemen zu generieren, ist es notwendig, Zufallszahlen mittels eines entsprechenden Zufallszahlengenerator nachbilden zu lassen. Die Anzahl der Zufallszahlen, die erzeugt werden müssen, ist sehr groß. Für die Simulation von 10 Produkten und 20 Kunden, die im Zeitraum zwischen 10 und 20 kaufen, müssen über 1.800 Zufallszahlen erzeugt werden. Erhöht man die Anzahl der Kunden auf 200, so werden bereits mehr als 19.000 Zufallszahlen erzeugt. Die Methode zur Gewinnung der Zufallszahlen ist daher von großer Bedeutung für den Erfolg des Zufallszahlengenerators.

Bei der Erzeugung von Zufallszahlen unterscheidet man zwischen echten Zufallszahlen, die im Zuge von Zufallsexperimenten (z.B. Glückspiel) erzeugt werden und Pseudozufallszahlen, die durch spezielle deterministische Algorithmen erzeugt werden und damit reproduzierbar sind [RECHE00].

5.2.1 Echte Zufallszahlengeneratoren

Bei echten Zufallszahlengeneratoren handelt es sich um mechanische oder elektrische Geräte, die Zufallszahlen aus der ihnen zugrunde liegenden Verteilung

liefern. Derartige Generatoren existieren bereits sehr lange. Ein Beispiel für einen mechanischen Zufallszahlengenerator ist der Würfel [Trenkl98].

Ein Beispiel für einen elektrischen Zufallszahlengenerator ist der Generator ERNIE(Electronic Random Number Indicator Equipment). Bei diesem Generator werden Geräuschimpuls von Neonröhren innerhalb konstanter Intervalle in Zufallszahlen umgewandelt werden. Verwendet wurde dieser Generator u.a. bei Lotto-Ziehung in Großbritannien [Trenkl98].

Der Vorteil von ,,echten" Zufallszahlengeneratoren ist, dass der Ausgang des Experiments nicht vorhersehbar ist. Dies ist aber gleichzeitig auch ein entscheidender Nachteil, da es nicht möglich ist, ein zweites Mal ein Experiment mit genau den gleichen Zufallszahlen durchzuführen. Wie [Trenkl98] ausführt ist „ein Grund für ein solches Vorgehen z.B. die Veränderung von Parametern der zugrunde liegenden Verteilung, um Veränderungen im Simulationsablauf vorher und nachher festzustellen“.

Ein weiterer Nachteil der echten Zufallszahlengeneratoren ist, dass sich aufgrund von Verschleiß der Generatoren die zugrunde gelegte Verteilung verändern kann. D.h. bei der Erzeugung von echten Zufallszahlen muss immer darauf geachtet werden, dass die Generatoren noch Zufallszahlen mit der zugrundegelegten Verteilung liefern [Trenkl98].

5.2.2 Pseudozufallszahlengeneratoren

Da die Beschaffung echter Zufallszahlen sehr langsam und aufwendig ist, versucht die Wissenschaft Zufallszahlen zu berechnen. Diese berechneten Zufallszahlen werden Pseudozufallszahlen genannt und mittels eines mathematischen Algorithmus erzeugt, der sich wie echte Zufallszahlengeneratoren verhalten sollen.

Da Pseudozufallszahlen aber mittels eines Algorithmus erzeugt werden, sind diese vollständig reproduzierbar, sodass mit Kenntnis dieses erzeugenden Algorithmus jeder die gleiche Folge von Zufallszahlen erzeugen bzw. voraussagen kann [Trenkl98].

[SuhlL00] fasst folgende Vor- und Nachteile der Pseudozufallszahlen zusammen:

Vorteile von Pseudozufallszahlen	Nachteile von Pseudozufallszahlen
• Schnell zu generieren • Volle Wiederholbarkeit • Keine externen Einflüsse • Günstige Hardware • Homogene Hardware • Testen im Voraus möglich	• Abhängigkeit von Startwerten • Perioden • Je nach Art des verwendeten Generators sehr unterschiedliche Güte der erzeugten Zahlenfolgen

Auch für den in dieser Arbeit entwickelten Testdatengenerator können keine echten Zufallsgeneratoren herangezogen werden, daher wird hier näher auf die Pseudozufallszahlengeneratoren eingegangen.

5.2.2.1 Die Quadratmitten oder Middle Square Method

Diese Methode wurde 1949 von J. Neumann vorgestellt. Es ist eine einfache Methode, bei der von einer Ausgangszahl das Quadrat gebildet wird. Die mittleren Ziffern dieser Zahl werden als erste Zufallszahl gewertet. Diese Zahl wird als Ausgangszahl für die nächste Zufallszahl genommen, d.h. sie wird wieder quadriert und die mittleren Ziffern ergeben die nächste Zufallszahl usw. so entsteht eine so genannte Zufallszahlenreihe. Die Ausgangszahl kann z.B. die Uhrzeit sein, oder die Anzahl an Millisekunden seit dem der Computer gestartet wurde [Trenkl98]. Knuth zeigte, dass sich nach dieser Methode, die Zufallszahlen nach 142 Zahlen wiederholen [Knuth96 S.4]. Aufgrund der vorher gezeigten Zahl an benötigten Zufallszahlen, scheidet diese Methode zur Bildung der Zufallszahlen für diesen Generator aus.

5.2.2.2 Der Lineare Kongruenzgenerator

Eine weitere häufig angewandte Methode ist die der „Linear Congruential Geneartors“ (LCG). LCGs sind in folgender Form aufgebaut [MoBy84]:

$(ax_n + b) \% m$

x_n ist hierbei die die zuletzt gebildete Zufallszahl und a, b und m sind Parameter. Wenn modulus der Word-Größe des Prozessors entspricht und die Berechnung in unsigned Int durchgeführt wird, reduziert sich die Formel auf:

$(ax_n + b)$

da mit dem Überlauf der Speicherstellen gearbeitet wird.

Diese Formel zeigt jedoch, dass die Zufallszahlenreihe von den Parametern a,b und m abhängt und reproduzierbar ist. D.h. die gleichen Parameterwerte führen zu der selben Zahlenreihe. Wenn die drei Parameter sinnvoll gewählt werden, ist die Periode der Zahlenreihe, d.h. die Anzahl an Werten nachdem sich die Werte wiederholen, ausreichend lang um von Zufallszahlen sprechen zu können. Da nach der zweiten Formel nur eine Multiplikation und eine Addition notwendig sind, können die Zufallszahlen sehr schnell berechnet werden.

Um eine maximale Periode zu erhalten sind nach [MoBy84] p.58 folgende Regeln bei der Auswahl der Parameter zu beachten:

1. b und m dürfen keine Faktoren außer 1 gemeinsam haben
2. (a-1) ist ein Vielfaches jeder Primzahl, die m dividiert
3. (a-1) ist ein Vielfaches von 4, wenn m ein Vielfaches von 4 ist

Wählt man die Parameter a,b und m so, dass eine maximal lange Periode entsteht, erhält man Zahlenfolgen, die nicht als Zufallsreihen zu klassifizieren sind. Setzt man a = b = 1, so erhält man folgende Zahlenreihe: (1,2,3,4,5,...). Somit hat man die längstmögliche Periode, jedoch keine Zufallszahlen da auf den ersten Blick die nächste Zahl vorhergesagt werden kann.

Beispiel für einen Kongruenzgenerator mit maximaler Periode:

a=5; b=1, m=16; $x_0 = 0$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_n	0	1	6	15	12	13	2	11	8	9	14	7	4	5	10	3	0

Für diese Wahl der Parameter sind die 3 Regeln erfüllt:

Ad 1: Primfaktorenzerlegung für b und m:

b = 1 * 1

m = 2 * 2 * 2 * 2 * 1

b und m haben keinen gemeinsamen Faktor (außer 1)

Ad 2: (a-1)=4

Die einzige Primzahl, die m dividiert ist die 2 und 4 ist ein Vielfaches von 2.

Ad 3: (a-1) = 4 (somit ein Vielfaches von 4)

m ist ein Vielfaches von 4, da 4*4 = 16

Erzeugt man mit diesen Werten 100 Zufallszahlen und trägt diese in einem Diagramm auf, erhält man folgenden Plot:

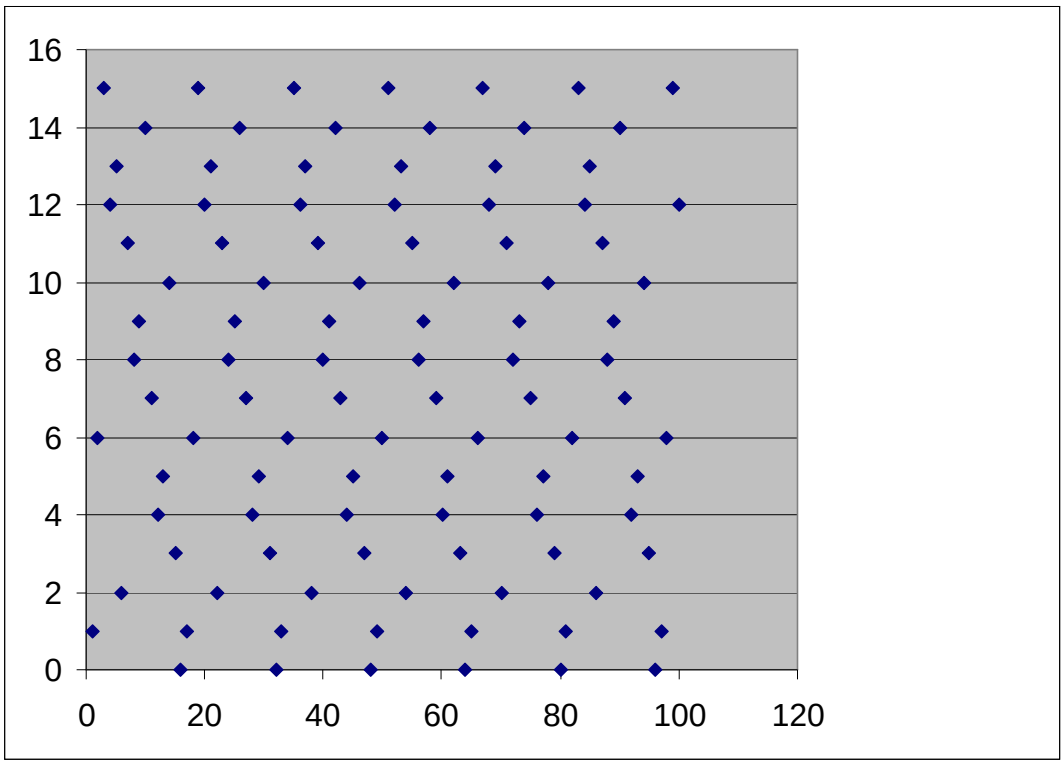


Abb: 5 Einfacher Kongruenzgenerator

Man erkennt hier ein eindeutiges Muster, da mit diesen Werten nur eine Periode von m = 16 möglich ist und sich danach die Zahlen wiederholen. Anhand dieses Musters ist es möglich Vorhersagen für die nächste „Zufallszahl“ zu treffen.

Wählt man die Werte a=781, b=387 und m =1000 erhält man folgende Zufallszahlen:

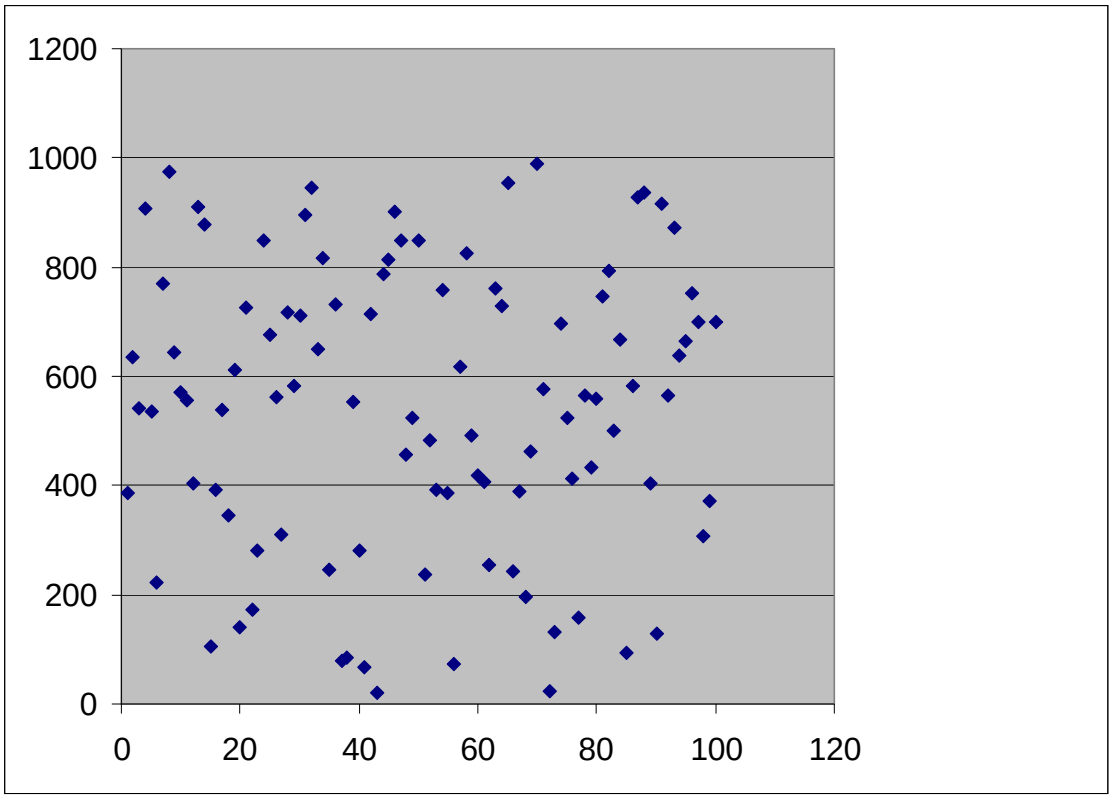


Abb: 6 Großer Kongruenzgenerator

Hier wirkt das Ergebnis zufälliger, jedoch kann auch hier ein Muster aufgezeigt werden. Es ist jedoch noch nicht zu erkennen, da nur 100 Zahlen gebildet wurden und sich die Zahlen erst nach 1000 Zahlen wiederholen. Für eine geringe Menge an Zufallszahlen wäre dieser Algorithmus ausreichend.

[SuhlL00] sieht folgende Vor- und Nachteile bei dieser Methode:

Vorteile der Linearen Kongruenzgeneratoren	Nachteile der Linearen Kongruenzgeneratoren
• einfach in der Konstruktion • benötigen wenig Speicher • schnell • liefern relativ gute Zahlenfolgen • Eigenschaften sind gut untersucht • Es gibt genaue Kriterien zur Parameterwahl • Durch diese Parameter kann man die Eigenschaften vorhersagen	• Bei der Wahl falscher Startparameter ist die Verteilung der Zahlen schlecht • Bei kommerziellen Anwendungen ist es schwierig, zufällige Startwerte zu generieren

Aus Sicht von [SuhlL00] überwiegen aber die Vorteile die Nachteile. Dies wird auch dadurch bestätigt, dass dieser Generator in der Praxis durchaus weit verbreitet ist.

5.2.2.3 Mersenne Twister

In den meisten Simulationsprogrammen werden jedoch mehr Zufallszahlen benötigt als mit den LCGs erzeugt werden können. Daher wurde von Makoto Matsumoto und Takuji Nishimura an der Keio Universität in Japan ein anderer Ansatz gewählt. Die beiden Autoren nennen Ihren Algorithmus „Mersenne Twister“ oder kurz „MT19937“.

Dieser Algorithmus wurde 1998 in [MatNish98] veröffentlicht und seither vielfach bei Simulationen eingesetzt. Der Generator hat alle gängigen Tests bisher bestanden darunter sind insb. der Spectral-Test nach [Knuth81], der k-distribution Test der in [MatNish98] beschrieben wird und der „die Hard“ Test [Marsa02].

Der Name MT19937 beinhaltet die Besonderheit dieses Generators. Die Periode beträgt 2^{19937} -1. Bei den LCG hängt die Periode mit der Prozessorgröße zusammen. Beim MT ist diese Periode auf allen Systemen zu erreichen. Im Dezimal-System entspricht die Zahl ca. 10^{6000} Leichter erkennt man die Größe der Zahl wenn man bedenkt, dass ca. 2^{80} Microsekunden die Zeitspanne von 10 Milliarden Jahren

bedeutet. Diese Menge an Zahlen bedeutet in dem Fall des Testdatengenerators ausreichende Zufallszahlen.

Der MT arbeitet mit 623 32-Bit Werten. Diese werden zunächst mittels eines LCG bestimmt. Die Zufallszahlen ergeben sich durch Bit-Austausch (Twist) innerhalb dieser 623 Zahlen.

Ein vereinfachtes Beispiel mit 3 Dezimalzahlen veranschaulicht den Prozess:

Zunächst werden 3 Zufallszahlen bestimmt:
123
221
332
Um nun neue Zahlen zu bestimmen, nimmt man die erste Ziffer eines Wertes und die letzten zwei Ziffern der nächsten Zahl. Somit erhält man nun:
121
232
323

Da dieser Generator mit Bit Operationen arbeitet, ist er sehr schnell, wie ein Vergleich in [MatNish98] der auf einer Sun Workstation durchgeführt wurde zeigt. Dabei wurden 10^7 Zahlen erzeugt. Die Zeiten der getesteten Generatoren lagen zwischen 9,24 Sekunden und 23,23 Sekunden, wobei MT die Aufgabe in 10,18 Sekunden bewältigte und mit Abstand die längste Periode hat.

5.2.3 Umrechnung in andere Verteilungen

Alle vorgestellten Zufallszahlengeneratoren liefern gleichverteilte Zahlen. Für den Testdatengenerator der in dieser Arbeit erstellt wird, werden jedoch Exponentialverteilte und Gammaverteilte Zufallszahlen benötigt.

5.2.3.1 Exponentialverteilung

Um aus einer gleichverteilten Zufallszahl u eine exponentialverteilte Zufallszahl x zu berechnen, wird folgende Formel angewendet:

$x = (-\ln(1-u)) / m$

wobei m mean der Exponentialverteilung ist.

In dem Testdatengenerator kommt ein Mersenne Twister zum Einsatz um gleichverteilte Zufallszahlen zu erzeugen und die oben angeführte Formel erzeugt die Exponentialverteilung. Berechnet man 1.000.000 Zahlen mit einem Mean von 0,1 und trägt diese in einem Diagramm ein, erhält man folgendes Bild:

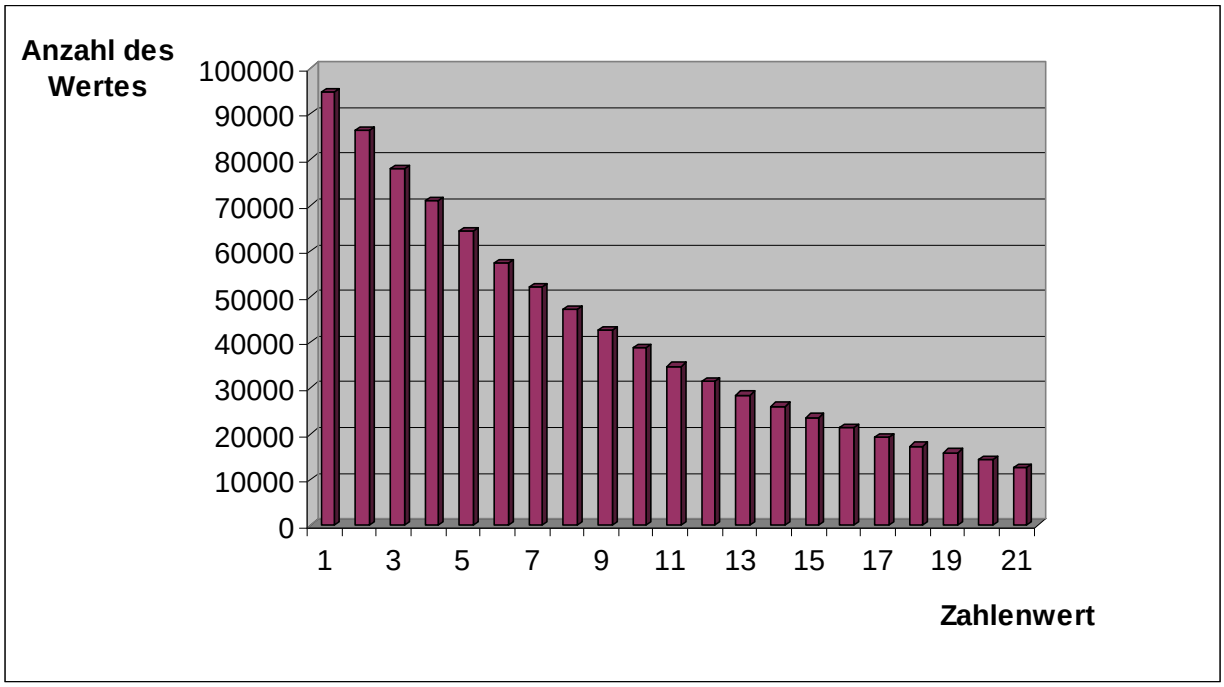


Abb: 7 erzeugte Exponentialverteilung

5.2.3.2 Gammaverteilung

Um Gamma-verteilte Zufallszahlen zu erzeugen, gibt es keine Transformationsformel. Diese Berechnung ist etwas aufwendiger. Knuth stellt in [Knuth81 S. 129] einen Algorithmus vor, um aus gleich verteilten Zufallszahlen Gamma-verteilte zu erzeugen. Im Testdatengenerator wird eine Implementation dieses Algorithmus verwendet die in [NRINC92] vorgestellt wird. Diese Implementation vermeidet die Berechnung des Tangens um mit allen Systemen kompatibel zu bleiben und die Effizienz des Algorithmus zu erhöhen.

Der ursprünglich von Knuth vorgestellte Algorithmus:

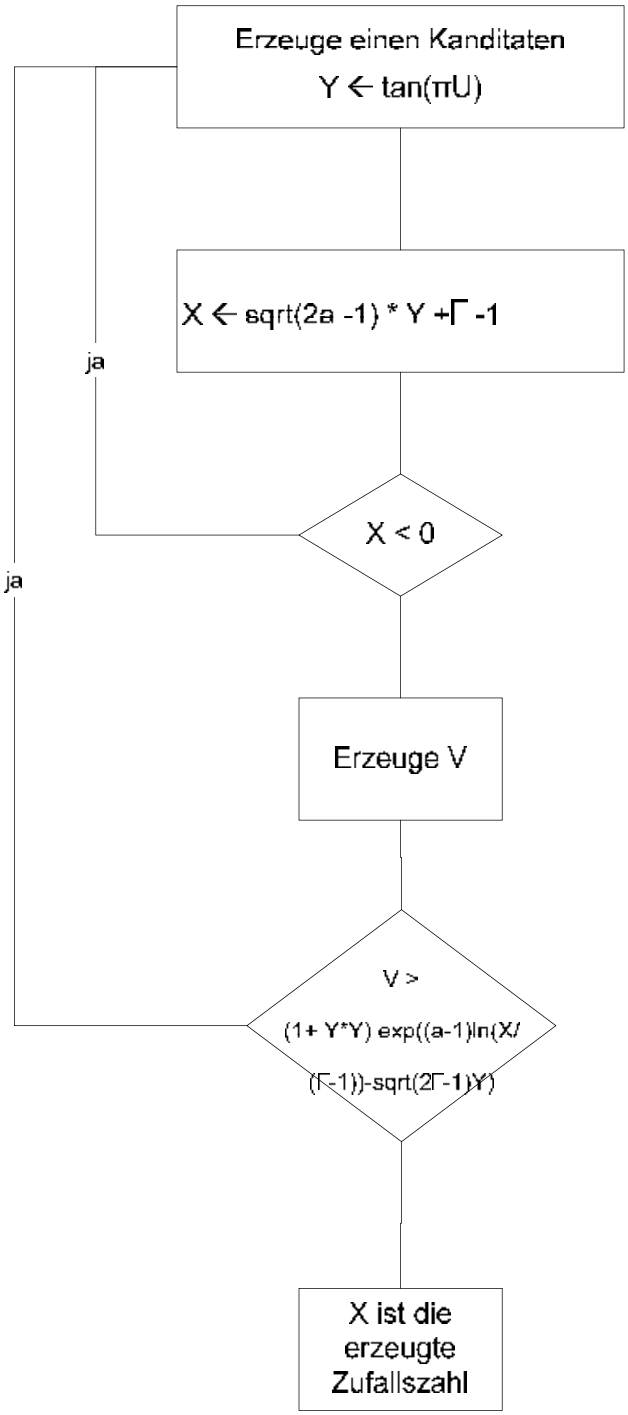


Abb: 8 Algorithmus für Gammaverteilung

U und V sind gleich verteilte Zufallszahlen, die vom Mersenne Twister erzeugt werden. ? ist der Parameter Gamma der Verteilung. Die ersten beiden Schritte des Algorithmus werden solange durchgeführt, bis das Ergebnis der Berechnung für X größer oder gleich 0 ist. Danach wird eine zweite gleich verteilte Zufallszahl erzeugt (V). Sollte der Ausdruck exp((? -1)ln(X/(? -1))-sqrt(2 ? -1)Y) kleiner V sein, wird der Algorithmus von vorne begonnen. Andernfalls ist X die erzeugte Zufallszahl.

Laut Knuth wird der Algorithmus durchschnittlich weniger als 1.902mal durchgeführt für Zufallszahlen mit einem Gamma > 3. Die verwendeten Funktionen (tan, sqrt, exp und ln) sind sehr rechenintensive Funktionen. Daher ist die Berechnung einer Gamma-verteilten Zufallszahl langsam. Eine Verbesserung wird dadurch erwirkt, dass anstelle der Tangensfunktion eine alternative Berechnung des Tanges durchgeführt wird, die nur Multiplikationen benötigt.

Erzeugt man mit dem Testdatengenerator der den vorgestellten Algorithmus verwendet 1.000.000 Zahlen und trägt diese in einem Diagramm auf, erhält man folgendes Bild der Verteilung, wobei Gamma = 4 verwendet wurde:

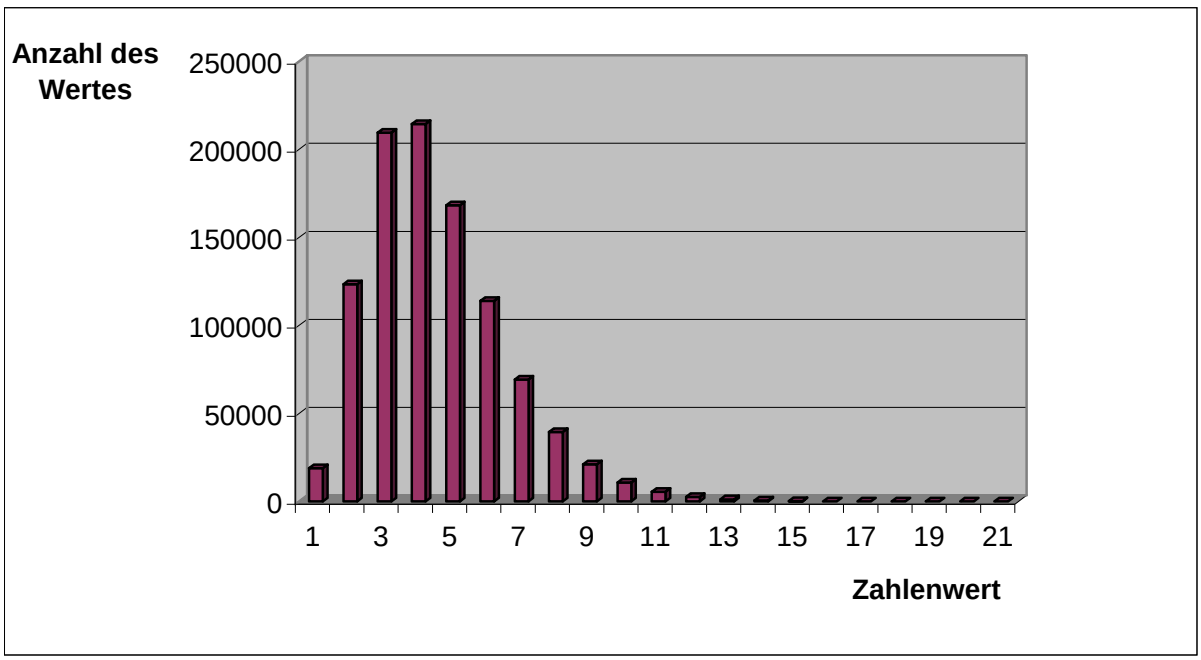


Abb: 9 Erzeugte Gammaverteilung

6 Dokumentation des Programms

6.1 Kompilieren des Programms

Das Programm wurde in Ansi C++ geschrieben und kann daher mit jedem kompatiblem C++ Compiler übersetzt werden. Der Source Code ist in zwei Dateien aufgeteilt (TDG.h und TDG.cpp)

Es wird die Standard Template Library verwendet. Somit muss diese beim kompilieren angegeben werden. Für den gcc Compiler unter CygWin lautet der Compile-Aufruf:

```
gcc tdg.cpp –o tdg.exe –lstdc++
```

6.2 Benutzerdokumentation

Das Programm ist als Kommandozeilenprogramm ausgelegt. Mit Parametern die beim Start angegeben werden, können die erzeugten Daten verändert werden. Die Ausgabedatei kann in zwei verschiedenen Formaten ausgegeben werden. Ein Format erzeugt eine Transaktion pro Zeile. Das zweite Ausgabeformat schreibt jede jedes gekauft Produkt in eine eigene Zeile. Die Transaktion ergibt sich dann aus den übereinstimmenden Zeitpunkten. Näheres über die Ausgabeformate findet man unter „6.3.3.2 Datendatei“.

6.2.1 Startparameter

-h: dieser Parameter listet eine Übersicht über die möglichen Startparameter auf.

-c: Der Parameter c gibt an, mit wie vielen Kunden das Programm arbeiten soll. Nach dem Buchstaben c folgt die Anzahl. Bsp: `-c 100` Das Programm erzeugt 100 Kunden.

-p: Mit p wird die Anzahl der Produkte festgelegt aus denen jeder Kunde wählen kann. Bsp: `-p 40` Jeder Kunde kann aus 40 Produkten wählen.

-m: Jeder Kunde kauft unterschiedlich oft ein Produkt. Jedoch ist die Anzahl der Käufe je Kunde über die Menge der Kunden gammaverteilt mit einem Parameter m. Nicht jedes Produkt hat denselben Parameter m. Daher wird beim Programmaufruf

ein Band festgelegt, innerhalb dessen die Produkte ihren Parameter m bestimmen. Bsp: `-m 0 4` bedeutet, dass jedes Produkt einen Parameter m zwischen 0 und 4 hat, mit dem das Programm rechnet.

-o: Hinter diesem Parameter wird der Dateiname der Ausgabedatei angegeben. Wobei eine Datei mit der Endung .txt erstellt wird. Diese Datei beinhaltet die Daten. Die zweite Datei mit der Endung .html enthält im HTML-Format nähere Informationen zu den erzeugten Daten. Es werden die Startparameter ausgegeben, sowie von jedem erzeugten Produkt, der verwendete Parameter m.

-tlist: Dieser Paramter wird als Switch verwendet. Wird –tlist beim Aufruf des Programmes angegeben wird die Ausgabedatei in der Form geschrieben, dass jede Transaktion nur eine Zeile in der Ausgabendatei benötigt. (siehe: 6.3.3.2.1 Format: Transaktionsliste)

Wenn –tlist nicht angegeben wird, kommt das „POS“-Ausgabeformat zur Anwendung. Dabei wird jede Transaktion in mehreren Zeilen in die Datei geschrieben. (siehe: 6.3.3.2.2 Format: „POS“-Liste)

-t: Hiermit wird festgelegt welchen Zeitraum die Kunden als Transaktionszeitraum verwenden sollen. Bsp: -t 0.01 bedeutet für einen Kunden, dass alle Kaufwünsche zwischen t und t+0.01 als eine Transaktion betrachtet werden.

6.2.2 Bemerkungen

Eine vorhandene Ausgabedatei wird überschrieben. Die Berechnung der Kaufwünsche und Transaktionen kann einige Zeit in Anspruch nehmen. Bei der Ausführung des Programms werden die einzelnen Aktionen am Bildschirm ausgegeben, um den Fortschritt kontrollieren zu können. Das Programm endet ohne besondere Meldung.

6.3 Systemdokumentation

6.3.1 Überblick über das Systemdesign

6.3.1.1 Systemspezifikation

Aufgabenstellung für die Entwicklung des Programms sind die in „4. Berechnung der Transaktionen “ beschriebenen Regeln für die Bildung von Transaktionen in ein C++ Programm umzusetzen.

Dieses Programm muss unter Linux kompilierbar sein und daher wurde nur Ansi C++ und die STL verwendet.

6.3.1.2 Klassendiagramm

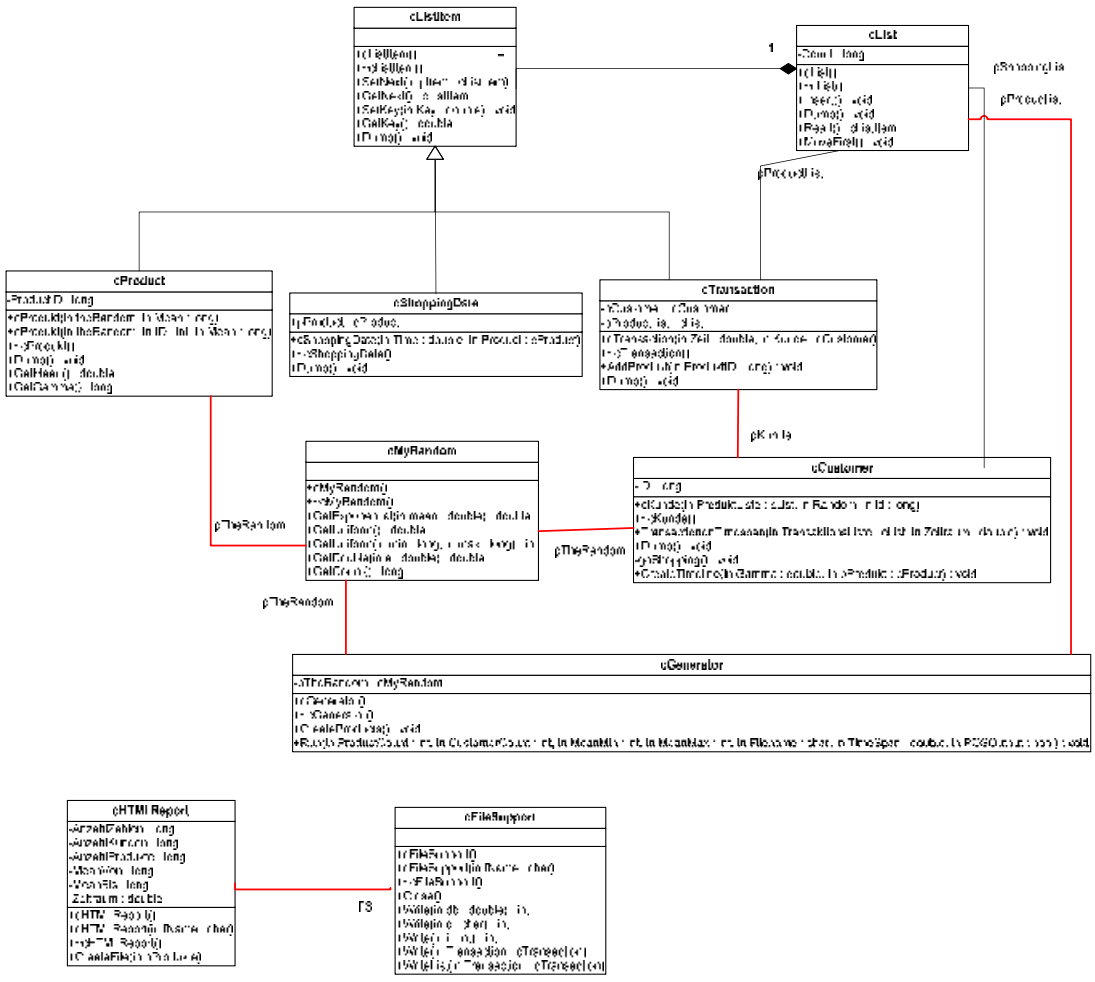


Abb: 10 Klassendiagramm

6.3.1.3 Beschreibung der Klassen

Name:	cFileSupport
Aufgabe:	Stellt alle notwendigen Funktionen und Methoden bereit um Informationen in eine Datei zu schreiben.
Basisklasse:	Keine
Benötigte Klassen:	Keine

Öffentliche Schnittstelle:	`cFileSupport(void);`
	Standardkonstruktor der Klasse. Als Dateiname wird “output.txt” verwendet.
	
	`cFileSupport(char* fName);`
	Ein weiterer Konstruktor. Hierbei wird der Dateiname der neuen Datei angegeben. Bei beiden Konstruktoren ist darauf zu achten, dass eine gegebenenfalls vorhandene Datei überschrieben wird.
	
	`~cFileSupport(void);`
	Destruktor der Klasse. Hier wird die Datei geschlossen. In einer abgeleiteten Klasse sollte dieser Destruktor aufgerufen werden.
	
	`void Close(void);`
	In dieser Methode wird die Datei geschlossen.
	
	`int Write(double db);`
	`int Write(char* c);`
	`int Write(int i);`
	Methoden um die jeweiligen Datentypen in einer eigenen Zeile in die Datei auszugeben.
	
	`int Write(cTransaktion* Transaktion);`
	Eine Methode um ein Objekt vom Typ cTransaktion in einer Zeile in der Datei darzustellen.
	
	`int WriteList(cTransaktion* Transaction);`
	Eine Methode, die ein Objekt vom Typ cTransaktion in mehreren Zeilen in die Datei schreibt.

Name:	cHTMLReport
Aufgabe:	Nach jedem Durchlauf erstellt das Programm eine HTML-Datei mit den wichtigsten Daten. Diese Datei wird mit dieser Klasse erstellt.
Basisklasse:	Keine
Benötigte Klassen:	cList, cFileSupport, cProdukt

Öffentliche Schnittstelle:	`cHTMLReport(void);`
	Der Default-Konstruktur erzeugt die Html-Datei mit dem Namen “output.html”
	
	`cHTMLReport(char* fName);`
	Dieser Konstruktor erlaubt es, den Dateinamen für die Ausgabedatei mit anzugeben.
	
	Bei beiden Konstruktoren wird ein cFileSupport Objekt angelegt.
	
	`~cHTMLReport(void);`
	Das cFileSupport Objekt wird hier mittels delete aus dem Speicher gelöscht.
	
	`void CreateFile(cList* pProdukte);`
	Diese Methode schreibt alle Daten in das Html-File. Es wird mit Write-Methoden purer HTML-Code in die Textdatei geschrieben. Am Seitenanfang werden die folgenden Parameter ausgegeben um nachvollziehen zu können, mit welchen Parametern das Programm gestartet wurde.
	
	Die übergebene Liste dient dazu, von jedem Produkt das verwendete Gamma in einer Liste in der HTML-Datei ausgeben zu können.
	
	Parameter die geschrieben werden:
	`long CountFigures;`
	`long CountCustomers;`
	`long CountProducts;`
	`long MeanMin;`
	`long MeanMax;`
	`double TimeSpan;`

Name:	cList
Aufgabe:	Es wurde für die Datenhaltung im Hauptspeicher eine eigene einfach verkettete Liste implementiert, die nur einen sequentiellen Zugriff auf die enthaltenen Elemente bietet. Die Elemente die in die Liste gespeichert werden können, sind vom Typ cListItem.
Basisklasse:	Keine
Benötigte Klassen:	cListItem

Öffentliche Schnittstelle:	`cList(void);`
	Der Konstruktor setzt vom ersten Element in der Liste, den Nachfolger auf das letzte Element, welches ebenfalls in der Liste gespeichert wird.
	
	`~cList(void);`
	Der Destruktor löscht alle Objekte, die in der Liste gespeichert sind mittels “delete” aus dem Speicher.
	
	`void Insert(cListItem* pItem);`
	Diese Methode fügt das übergebene Element an der richtigen Position in die Liste ein. Ein cListItem-Objekt liefert mit der Methode GetKey() einen double Wert, der hier als Sortierkriterium herangezogen wird.
	
	`void Dump();`
	Ist eine Hilfsfunktion, die den Inhalt der Liste auf der Konsole ausgibt. Diese Methode ruft von jedem cListItem die Methode Dump() auf.
	
	`cListItem* Read();`
	Die Read() Funktion liefert ein cListItem oder null. Wenn ein cListItem zurückgegeben wird, kann die Funktion ein weiteres mal aufgerufen werden und man erhält das nächste Item. Diese Funktion wird verwendet, um durch die Liste zu navigieren. Wenn null übergeben wird, bedeutet dies, am Ende angelangt zu sein.
	
	`void MoveFirst(void);`
	Mit dieser Methode wird beim nächsten Read()-Aufruf wieder am Anfang der Liste begonnen.
	
	`long Count;`
	Enthält die aktuelle Anzahl an Elementen in der Liste.

Name:	cListItem
Aufgabe:	Diese Klasse dient dazu, Elemente in cList abspeichern zu können. Meistens wird diese Klasse als Basisklasse für andere Elemente verwendet.
Basisklasse:	Keine
Benötigte Klassen:	Keine

Öffentliche Schnittstelle:	`cListItem(void);`
	Der Konstruktor initialisiert den Zeiger auf das folgenden Element mit NULL
	
	`~cListItem(void);`
	Ohne Funktion
	
	`void SetNext(cListItem* pItem);`
	Hiermit kann das Nachfolgeelement gesetzt werden. Dazu wird ein Zeiger auf dieses Element übergeben.
	
	`cListItem* GetNext(void);`
	Diese Funktion liefert den Zeiger auf das Nachfolgeelement.
	
	`void SetKey(double Key);`
	Jedes Element hat einen Key der aus einer Doublezahl besteht. Die Keys aller Elemente in einer Liste müssen nicht eindeutig sein. Sie dienen der richtigen Einsortierung in der Liste.
	Der Doubledatentyp wurde gewählt, da die häufigste Anwendung das Speichern von Zeitpunkten ist. In diesem Programm wird die Zeit im Intervall 0..1 dargestellt. Somit sind die Punkte auf der Zeitachse als Double-Zahl darstellbar.
	
	`double GetKey();`
	Liefert den Key des Elements zurück.
	
	`virtual void Dump();.`
	Diese Funktion sollte von den abgeleiteten Klassen überschrieben werden und stellt das Objekt in der Konsole dar. In der cListItem-Klasse wird der Inhalt des Key ausgegeben.

Name:	cProduct
Aufgabe:	Alle Produkte die ein Kunde kaufen kann, werden als eine Objekte im Speicher gehalten. Diese Klasse dient dazu das Produkt zu speichern.
Basisklasse:	cListItem
Benötigte Klassen:	cMyRandom

Öffentliche Schnittstelle:	`cProduct(cMyRandom* theRandom, long Mean);`
	`cProduct(cMyRandom* theRandom, int ID, long Mean);`
	Beide Konstruktoren erhalten einen Zeiger auf die Instanz des Zufallszahlengenerators übergeben. Weiters wird ein Mean übergeben. Jeder Kunde hat für jedes Produkt ein unterschiedliches Kaufverhalten. Das Produkt gibt dem Kunden diesen Wert in einer Funktion zurück. Alle Werte die das Produkt liefert sind um diesen Mean-Wert Gamma-verteilt.
	Wird der Konstruktor ohne ID verwendet erhalt das Produkt den ID 0.
	
	`~cProdukt(void);`
	Keine Funktion
	
	`void Dump();`
	Überschrieben von cListItem und gibt von dem Produkt den ID auf der Konsole aus.
	
	`double GetMean();`
	Liefert den beim Konstruktor übergebenen Mean-Wert zurück.
	
	`long GetGamma();`
	Bei jedem Aufruf dieser Funktion wird vom Zufallszahlengenerator eine neue, Gamma-verteilte Zufallszahl berechnet und zurückgegeben.
	
	`long ProductID;`
	Eine öffentlich zugängliche Variable, die den ProduktID enthält.

Name:	cCustomer
Aufgabe:	Jeder Kunde wird als eigenes Objekt im Speicher gehalten. Dazu dient diese Klasse. Außerdem simuliert die Klasse den Einkaufsprozess des Kunden für jedes Produkt.
Basisklasse:	Keine
Benötigte Klassen:	cMyRandom, cList, cProduct, cShoppingDatet

Öffentliche Schnittstelle:	`cKunde(cList* ProductList, cMyRandom* Random, long id);`
	Beim Erzeugen des Kunden wird der ID des Kunden und ein Zeiger auf den allgemeinen Zufallszahlengenerator übergeben.
	Außerdem wird ein Zeiger auf die Produktliste übergeben. Der Kunde kann aus diesen Produkten bei seinem Einkauf wählen.
	
	`~cKunde(void);`
	Keine Funktion
	
	`void TransactioneTimespan(cList* TransactionsList, double TimeSpan);`
	Mit dieser Funktion werden aus den einzelnen Bedarfszeitpunkten, die in der Kaufliste gespeichert sind Transaktionen gebildet.
	Eine Transaktion enthält alle Produkte die der Kunde in dem im Parameter „Zeitraum“ angegeben Zeitraum nachfragt.
	Jede Transaktion wird in der übergebenen TranksaktionsListe gespeichert und so zurückgegeben.
	
	`void Dump();`
	Diese Funktion gibt die Kaufliste des Kunden auf der Konsole aus.
	
	`long ID;`
	Eine öffentlich zugängliche Variable, die den KundenID enthält.

Anmerkungen:	`void goShopping(void);`
	In dieser Funktion wird für jedes Produkt in der Produktliste der Parameter Mean vom Produkt erzeugt. Und mit diesem Parameter wird der Poissionprozess durchgeführt.
	
	`void CreateTimeLine(double Gamma, cProduct* pProduct);`
	Diese Funktion erhält den Parameter Gamma und das Produkt aus der Funktion goShopping und erzeugt für den Kunden die einzelnen Zeitpunkte wann der Kunden einen Bedarf an diesem Produkt hat.
	Zu jedem Zeitpunkt wird eine Zeitspanne hinzugerechnet. Diese Zeitspanne ergibt sich aus einer exponential-verteilten Zufallszahl um den Parameter Gamma. Die Zeitpunkte werden solange gebildet bis der Zeitpunkt t=1 erreicht wird.
	Alle Zeitpunkte werden in eine Kaufliste des Kunden eingetragen.

Name:	cShoppingDate
Aufgabe:	Jede Nachfrage eines Kunden wird als Kaufzeitpunkt gespeichert. Es wird gespeichert welches Produkt zu welchem Zeitpunkt nachgefragt wird.
Basisklasse:	cListItem
Benötigte Klassen:	cProduct

Öffentliche Schnittstelle:	`cShoppingDate(double Time, cProdukt* Product);`
	Im Konstruktor wird der Zeitpunkt angegeben und ein Zeiger auf das Produkt, welches nachgefragt wird.
	Der Zeitpunkt ist vom Typ double und wird als Key der cListItem-Klasse gespeichert.
	
	`~cShoppingDate(void);`
	Keine Funktion
	
	`void Dump(void);`
	Ist von der Klasse cListItem überschrieben und gibt den Zeitpunkt und den ID des Produkts auf der Konsole aus.
	
	`cProduct* pProduct;`
	Eine öffentliche Membervariable die den Zeiger auf das Produkt enthält.

Name:	cTransaction
Aufgabe:	Mehrere Nachfragezeitpunkte eines Kunden werden zu einer Transaktion zusammengefasst. Diese Klasse stellt eine Transaktion im Speicher dar.
Basisklasse:	cListItem
Benötigte Klassen:	cList

Öffentliche Schnittstelle:	`cTransaction(double Time, long theCustomerID);`
	Im Konstruktor wird der Zeitpunkt der Transaktion und der Kunde dieser Transaktion angegeben. Der Zeitpunkt wird als Key der Basisklasse gespeichert.
	Außerdem wird eine interne Produktliste initialisiert.
	
	`~cTransaction(void);`
	Löscht die interne Produktliste mit delete.
	
	`void AddProduct(long ProductID);`
	Jede Transaktion enthält eine Liste an Produkten. Mit dieser Funktion wird ein Produkt zu der Liste hinzugefügt.
	
	`void Dump(void);`
	Überschrieben von cListItem und gibt den Zeitpunkt und Kunden der Transaktion aus. Danach wird die interne Produktliste ausgegeben.
	
	`cList* pProductList;`
	Eine intern gehaltene Liste, die alle Produkte, die in dieser Transaktion enthalten sind speichert.
	
	`long CustomerID;`
	Die ID des Kunden, der die Transaktion durchführt.

Name:	cMyRandom
Aufgabe:	Diese Klasse errechnet aus den gleichverteilten Zufallszahlen, die die Klasse TRandomMersenne liefert, Zufallszahlen die exponential oder gamma-verteilt sind.
Basisklasse:	Keine
Benötigte Klassen:	TRandomMersenne

Öffentliche Schnittstelle:	`cMyRandom();`
	Erzeugt ein neues TRandomMersenne Objekt.
	
	`~cMyRandom(void);`
	Zerstört das interne TRandomMersenne Objekt.
	
	`double GetExponential(double mean);`
	Diese Funktion erhält von TRandomMersenne eine gleich-verteilte Zufallszahl und errechnet mittels Transformationsfunktion eine exponential-verteilte Zufallszahl mit dem Parameter mean.
	
	`double GetUniform();`
	Gibt eine von TRandomMersenne erzeugte gleich-verteilte Zufallszahl ohne Veränderung zurück.
	
	`int GetUniform(long min, long max);`
	Gibt eine gleich-verteilte, ganzzahlige Zufallszahl im Bereich [min;max] zurück.
	
	`double GetGamma(double a);`
	Diese Funktion erzeugt nach dem Algorithmus von Knuth eine gamma-verteilte Zufallszahl mit dem Parameter a. Der verwendete Algorithmus wird in 5.2.3.2 Gammaverteilung beschrieben.
	
	`long GetCount();`
	Bei jeder von TRandomMersenne erzeugten Zufallszahl wird ein Counter erhöht und dieser wird von der Funktion zurückgeliefert. Dies schafft den Überblick über die benötigte Anzahl der Zufallszahlen, um bestimmen zu können, ob die Periode des verwendeten Zufallszahlengenerators ausreichend ist.

Name:	cGenerator
Aufgabe:	Der gesamte Programmablauf wird in dieser Klasse bewerkstelligt. Diese Klasse erhält die Ausgangsparameter und erzeugt die Ausgabedateien. Dadurch wird es möglich den Testdatengenerator in andere Programme einzubinden.
Basisklasse:	Keine
Benötigte Klassen:	Alle

Öffentliche Schnittstelle:	`cGenerator(void);`
	Hier wird das allgemeine cMyRandom-Objekt erzeugt, sowie die Transaktionsliste und die Produktliste.
	Weiters werden die Ausgangsparameter auf die Defaultwerte gesetzt.
	
	`~cGenerator(void);`
	Es werden das cMyRandomobjekt, die Transaktionsliste und die Produktliste mittels delete zerstört.
	
	`void CreateProducts(void);`
	Es werden die notwendige Anzahl an Produkten in die Produktliste aufgenommen. Für jedes Produkt wird eine gleich-verteilte Zufallszahl bestimmt, die in weiterer Folge das Kaufverhalten eines Kunden ausdrücken wird. Diese Zufallszahl ist vom Produkt mit der Funktion GetGamma() abzurufen.
	
	`void Run(int ProductCount,int CustomerCount, int MeanMin, int MeanMax, const char* Filename, double TimeSpan, bool POSOutput);`
	Diese Funktion enthält den eigentlichen Programmablauf. Die Parameter sind die Parameter, die die Erzeugung der Testdaten beeinflussen.
	In dieser Funktion wird zunächst die Produktliste erzeugt. Dann wird die notwendige Anzahl an Kunden erzeugt. Die Kunden erzeugen ihre Nachfragezeitpunkte und tragen die Transaktionen in die Transaktionsliste ein.
	Sobald dies für einen Kunden erledigt, werden die Zeilen dieser Transaktion in die Date geschrieben, und es wird beim nächsten Kunden fortgesetzt.
	Mit dem Parameter POSOutput wird das Ausgabeformat der Datendatei bestimmt.
	
	`cMyRandom* pTheRandom;`
	Die einzige Instanz des Zufallszahlengenerators.

6.3.1.4 Vorgangsweise bei der Berechnung

Die Methode Run() der Klasse Generator ist die eigentliche Hauptfunktion. In dieser Methode werden alle Schritte zur Berechnung durchgeführt. Alle Einstellungen für eine Berechnung, wie zum Beispiel Anzahl der Kunden oder Anzahl der Produkte werden dieser Methode als Parameter angegeben.

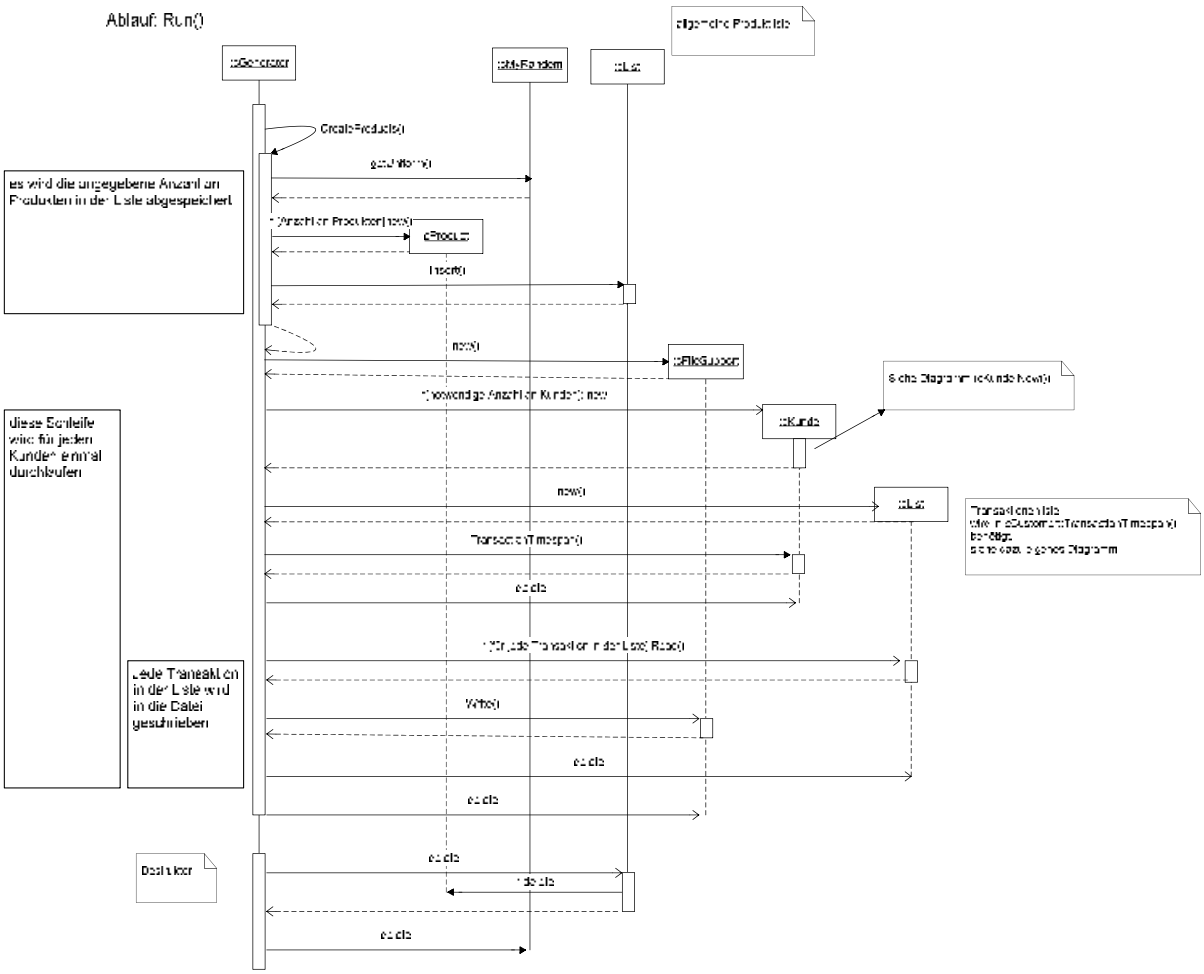


Abb: 11 Sequenzdiagramm: Run

Zunächst wird von dieser Methode die Methode CreateProducts() in der selben Klasse aufgerufen. CreateProducts() durchläuft eine Schleife um die angegebene Anzahl an Produkten in der Produktliste des Generators anzulegen. Bei jedem Schleifendurchlauf und somit für jedes Produkt wird aus der cMyRandom Klasse mittels getUniform() eine gleichverteilte Zufallszahl gezogen. Der hier gezogene Wert ist der Wert μ des Produktes.

Nach dem Aufruf der Methode createProducts() erfolgt die weitere Berechnung wieder in der Methode Run().

Es wird ein cFileSupport Objekt angelegt. Dieses Objekt ist für die weitere Ausgabe in die Outputdatei verantwortlich. Nachdem das Objekt angelegt wurde, beginnt eine Schleife, die entsprechend der Kundenanzahl ausgeführt wird.

Es wird eine Instanz der Klasse cCustomer angelegt. Im Konstruktor der Klasse werden die notwendigen Funktionen aufgerufen, um die Zeitpunkte der Einkäufe des Kunden festzulegen. Siehe dazu das eigene Diagramm cCustomer::New().

Danach wird eine Liste instanziert, die die Transaktionen des Kunden abspeichern wird. Durch den Aufruf von TransactionTimespan() in dem Kunden-Objekt werden aus den Zeitpunkten der Einkäufe Transaktionen gebildet und diese Transaktionen in der zuvor erzeugten Liste abgespeichert. Die Funktion TransactionTimespan() ist ebenfalls in einem eigenen Diagramm dargestellt. Nach dem Aufruf dieser Funktion wird das Kunden-Objekt aus dem Speicher entladen.

Nun werden die gebildeten Transaktionen aus der Liste gelesen und mittels cFileSupport-Objekt in eine Output-Datei geschrieben. Je nach dem gewählten Ausgabeformat wird die Funktion Write() oder WriteList() aufgerufen und diesen Funktionen eine Instanz einer Transaktion übergeben.

Nach dem Schreibvorgang wird das Listenobjekt mittels delete aus dem Speicher entfernt. Aufgrund der Implementation der Listenklasse werden die in der Liste abgespeicherten Objekte ebenfalls aus dem Speicher entladen.

Diese Schritte werden für jeden Kunden wiederholt. Nach dem letzten Kunden wird die Output-Datei geschlossen und das cFileSupport-Objekt aus dem Speicher entladen.

6.3.1.5 Sequenzdiagramm cCustomer:: New()

Der Ablauf im Konstruktor der Klasse cCustomer wurde in diesem Diagramm gesondert dargestellt um eine besser Übersichtlichkeit zu erreichen.

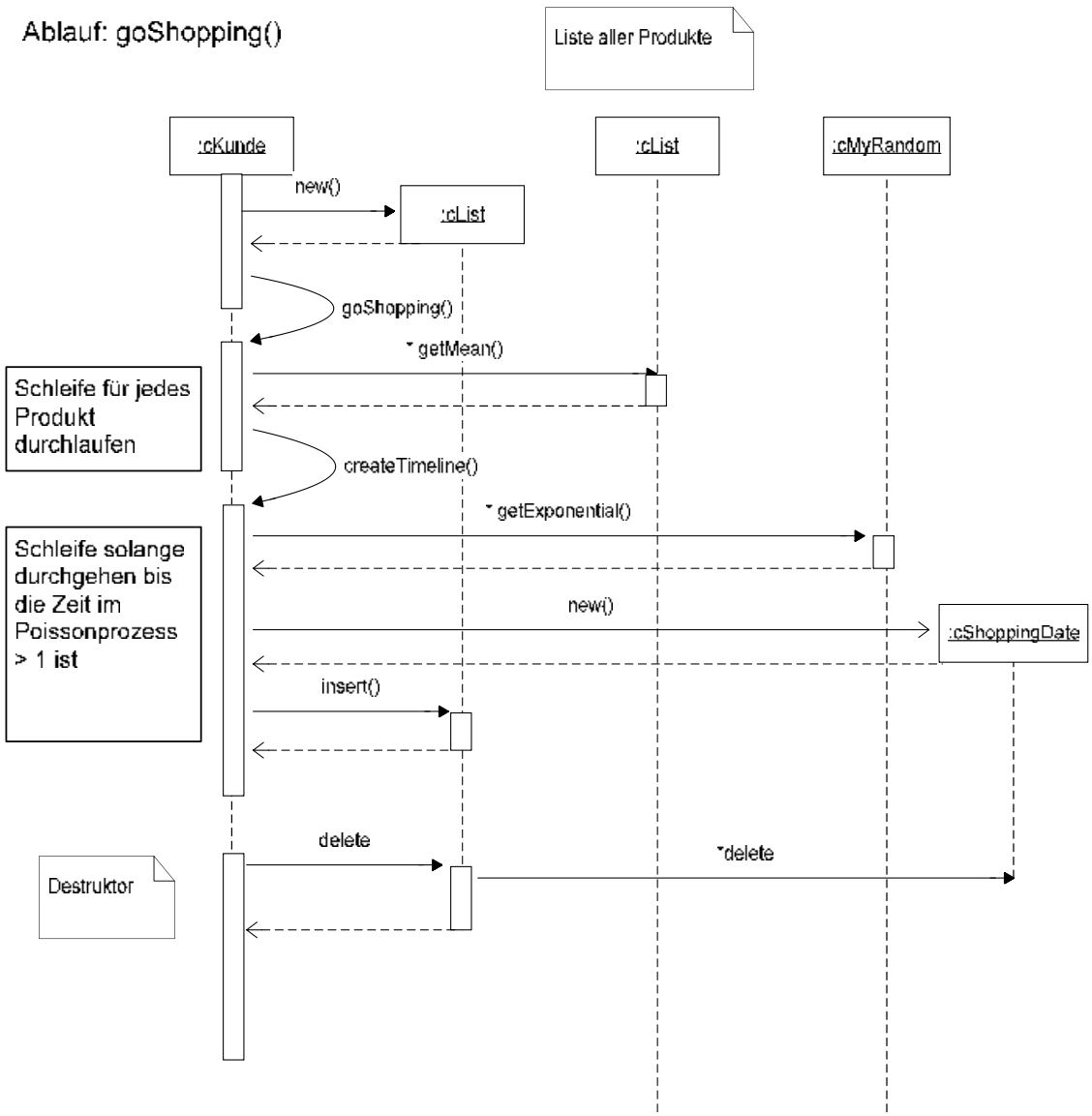


Abb: 12 Sequenzdiagramm: goShopping

Sobald ein Kundenobjekt instanziert wird, wird eine Liste angelegt, die die ermittelten Kaufzeitpunkte des Kunden aufnehmen wird (=Kunden-interne ShoppingList). Danach wird die Methode goShopping() aus der Klasse cCustomer aufgerufen.

Die Methode goShopping() besteht aus einer Schleife, die über alle Produkte, die in der allgemeinen Produktliste abgelegt sind iteriert. Zunächst wird von einem Produkt mit der Funktion getMean() ein Wert bestimmt. Dieser Wert ist über alle Käufer

dieses Produktes eine gamma-verteilte Zufallszahl um den Wert µ der bei der Erzeugung des Produktes festgelegt wurde. Mit diesem Parameter wird die Methode createTimeline() der Klasse cCustomer aufgerufen.

In createTimeline() werden die Zeitpunkte der Einkäufe des Kunden bezüglich des einen aktuellen Produktes bestimmt. Dies wird mittels Poisson-Prozess durchgeführt.

Es wird eine Schleife solange durchlaufen, bis die fiktive Zeit einen Wert größer als 1 erreicht. In dem Prozess wird bei jedem Schritt eine exponential-verteilte Zufallszahl mit dem Paramter µ zu der aktuellen Zeit hinzugezählt. Zu jeden so gefundenen Zeitpunkt wird eine Instanz von cShoppingDate erzeugt und der kunden-internen ShoppingList hinzugefügt.

Im Destruktor der Klasse cCustomer wird die interne ShoppingList gelöscht. Aufgrund der Implementation der cList Klasse werden auch alle cShoppingDate-Objekte in der Liste gelöscht.

6.3.1.6 Sequenzdiagramm cCustomer::TransactionTimespan

Nach dem Erzeugen eines Kunden-Objektes ruft die Run() Methode aus der Klasse cGenerator innerhalb der cCustomer Klasse die Methode TransactionTimespan() auf. Der Ablauf dieser Methode ist im folgenden Diagramm dargestellt.

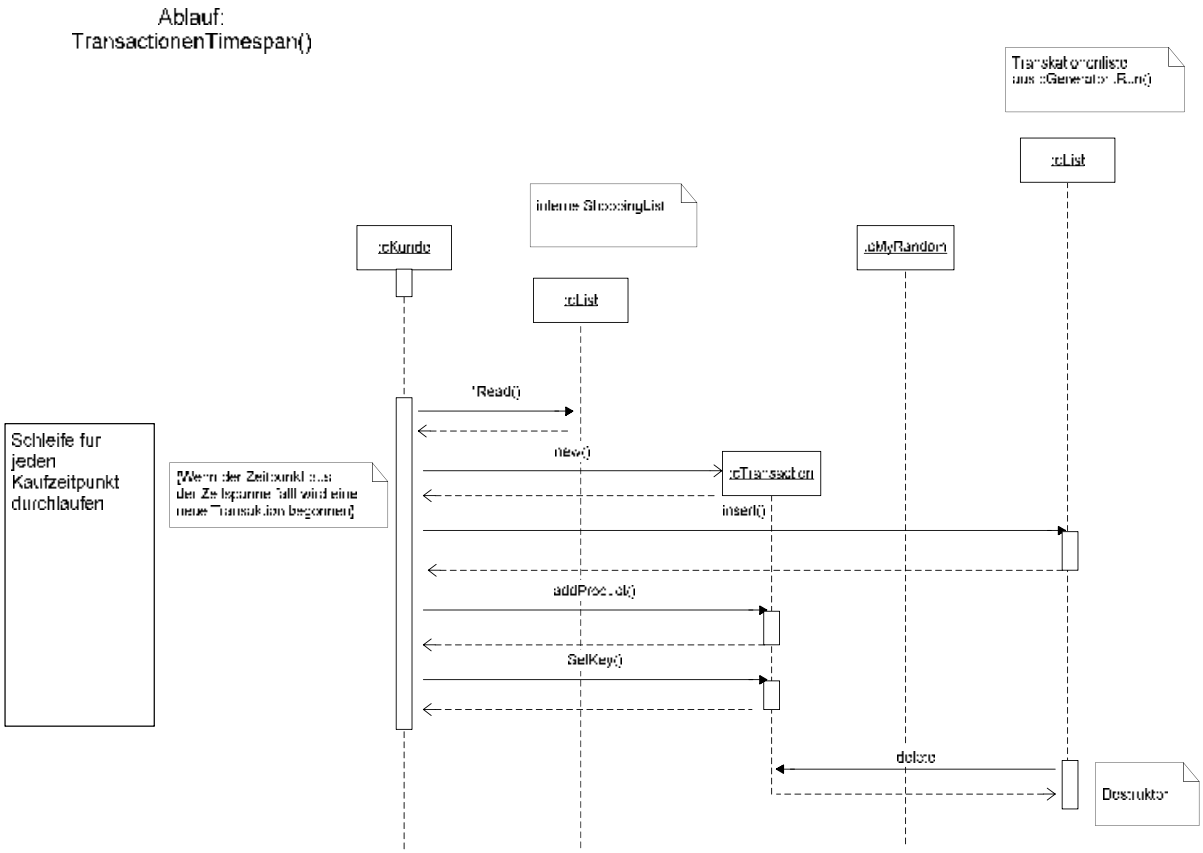


Abb: 13 Sequenzdiagramm TransactionenTimespan

Diese Methode erzeugt aus den zuvor gebildeten Kaufzeitpunkten eines Kunden einzelne Transaktionen die aus einer Sammlung an Produkten bestehen die innerhalb eines einstellbaren Zeitraumes liegen. Wobei der Zeitpunkt der Transaktion der Zeitpunkt des letzten Kaufzeitpunktes ist, der noch innerhalb dieser Transaktion liegt.

Es wird durch interne ShoppingList iteriert. Wenn der Zeitpunkt ausserhalb der Zeitspanne liegt, wird ein neues cTransaction Objekt erzeugt und in der Transaktionsliste abgespeichert. Ein Zeiger auf diese Liste wird beim Aufruf der Methode aus der Klasse cGenerator mit übergeben. Diese Liste wurde auch dort instanziert (siehe 6.3.1.4 Vorgangsweise bei der Berechnung)

In das aktuelle Transaction-Objekt wird mittes AddProduct() der ID des Produktes aus dem jeweiligen ShoppingTime-Objekt eingetragen. Danach wird mittels SetKey() der Zeitpunkt der Transaktion auf den Zeitpunkt des aktuellen ShoppingTime-Objektes gesetzt.

Im Destruktor der Transaktionenliste, der in der cGenerator::Run() Methode aufgerufen wird, werden alle erzeugten Transaktionen-Objekte gelöscht und aus dem Speicher entfernt.

6.3.1.7 Implementierung einer verketteten, sortierten Liste

Das Programm muss die verwendeten Produkte in einer Liste speichern. Ebenso werden die Kaufzeitpunkte und die Transaktionen in Listen gespeichert. In dem Testdatengenerator wurde eine eigene Liste in der Klasse cList programmiert.

Diese Liste kann Elemente vom Typ cListItem speichern. Jedes Element hat einen Key, der nicht eindeutig sein muss. Dieser Key ist vom Typ Double, da primär die Kaufzeitpunkte gespeichert werden müssen und die Zeit in diesem Programm im Zeitraum 0 bis 1 dargestellt wird. Beim Einfügen von Elementen in die Liste werden diese nach dem Key einsortiert.

Die Liste stellt eine Methode Dump() bereit um den Inhalt der Liste ausgeben zu können. Hierbei wird die Dump() Methode des cListItem aufgerufen. Um alle Elemente in der Liste gemäß der Sortierreihenfolge auslesen zu können, wurde die Methode GetNext() implementiert. Diese liefert das nächste Element oder Null, wenn keine weiteren Elemente zur Verfügung stehen.

Da die einzelnen Objekte die in Listen gespeichert werden müssen, wie Produkte, Kaufzeitpunkte und Transaktionen unterschiedliche Eigenschaften besitzen, werden die Klassen dieser Objekte von cListItem abgeleitet und können so in eine Liste vom Typ cList eingefügt werden.

Es wurde eine eigene Implementation einer Liste gewählt, da nur eine Methode zum Einfügen und eine Möglichkeit zur Iteration gegeben sein müssen, und kein zusätzlicher Overhead das Programm belasten soll.

6.3.1.8 Dateierzeugung

Um Daten in Dateien schreiben zu können wurde die Klasse cFileSupport programmiert. Dieser Klasse wird im Konstruktor der Dateiname der neuen Datei übergeben. Existiert diese Datei bereits, so wird sie überschrieben.

Weiters wurde eine Methode Write() implementiert. Diese wurde vierfach überladen, sodass es möglich ist, folgende Datentypen in die Datei zu schreiben: double, char*, int und die Klasse cTransaktion. Diese Methoden schreiben jeweils die übergebenen Daten in die Datei.

Abschließend muss die Methode close() aufgerufen werden um die Datei im Dateisystem zu schließen.

6.3.1.9 Kommandozeilenargumente

Um die vom Benutzer angegebenen Parameter beim Programmstart auslesen zu können wurde die Klasse CCmdLine verwendet [Animal99].

Einem Objekt dieser Klasse übergibt man die gesamte Kommandozeile. Nach dem Parsen der Zeile, wird über die Funktion GetArgument() der Wert eines Switches der Kommandzeile ausgelesen.

BSP: Parameter: -k 230

Die Funktion GetArgument(„k“) liefert den Wert „230“. Wird der Switch in der Kommandozeile nicht gefunden, so kommt es zu einer Fehlermeldung. Um das zu vermeiden existiert die Funktion GetSafeArgument() die im Programm verwendet wurde. Diese liefert keine Fehlermeldung sondern einen leeren String.

Diese Klasse verwendet die STL. Somit ist der Testdatengenerator mit der STL zu linken.

6.3.1.10 Zufallszahlenerzeugung

Kernklasse der Zufallszahlenerzeugung ist die Klasse TRandomMersenne. Diese Klasse implementiert einen Mersenne Twister nach [MatNish98]. Diese Klasse stellt Methoden bereit um Zufallszahlen vom Typ Double und Long zu erzeugen.

Die Klasse cMyRandom verwendet TRandomMersenne und transformiert die Gleichverteilten Zahlen des Mersenne Twister in die entsprechend verteilten Zufallszahlen.

Die Funktion GetUniform() liefert gleichverteilte Zufallszahlen, die direkt vom Mersenne Twister bezogen werden. Mit GetGamma(double) werden Gamma-

verteilte Zufallszahlen erzeugt. Der Algorithmus, der zur Berechnung verwendet wird wurde unter „5.1.2 Gammaverteilung“ beschrieben. Die Funktion GetExponential(double) errechnet eine exponential-verteilte Zufallszahl mittels der unter „5.1.1Exponentialverteilung“ beschriebenen Formel.

Die Klasse cMyRandom enthält eine statische Variable, die die Anzahl der berechneten Zufallszahlen zählt. Diese Anzahl dient der zusätzlichen Information über die Arbeitsweise des Programms und wird am Ende ausgegeben.

6.3.2 Implementierung im Detail

6.3.3 Beschreibung der Ausgabedateien

Das Programm erzeugt zwei Ausgabedateien. Eine mit den erzeugten Daten und eine Datei im HTML-Format, die die Ausgangsparameter enthält und als Dokumentation für die Erzeugung dienen kann.

6.3.3.1 HTML-Datei

Die Datei ist in zwei Bereiche geteilt. Im ersten Teil werden die Parameter aufgelistet mit denen das Programm gestartet wurde. Im zweiten Teil werden die erzeugten Produkte aufgelistet und von jedem Produkt der erzeugte Parameter Mean angegeben.

6.3.3.2 Datendatei

Die Datendatei kann in zwei verschiedenen Formaten erstellt werden. Das gewünschte Format ist beim Programmaufruf mittels Startparameter anzugeben.

6.3.3.2.1 Format: Transaktionsliste

Die Datendatei ist im Textformat erstellt und enthält die erzeugten Transaktionen wobei jede Zeile der Datei eine eigene Transaktion darstellt:

Die Zeilen sind folgendermaßen aufgebaut:

<TransaktionsID>;<Zeitpunkt>;<KundenID>;<Liste der ProduktIDs>

TransaktionsID: Jede Transaktion (Zeile) erhält eine fortlaufende, ganzzahlige ID

Zeitpunkt: Der Zeitpunkt der Transaktion als double Zahl im Bereich [0;1]

KundenID: Ganzzahlige ID des Kunden, der die Transaktion durchführt.

Liste der ProduktIDs: Hier werden die IDs der Produkte angeführt, die zu dieser Transaktion gehören.

Alle Werte sind durch Strichpunkt (;) getrennt und können so in weiteren Programmen ausgewertet werden.

Beispiel für einen Teil der Ausgabedatei:

```
00048;0.007974;00027;027;
00049;0.007996;00199;028;
00050;0.008004;00179;007;009;
00051;0.008016;00097;000;
00052;0.008180;00241;014;029;
00053;0.008353;00157;015;
00054;0.008375;00138;001;001;007;
```

Die Liste der Produkte in einer Transaktion kann eine beliebige Länge aufweisen. Dies ist in Auswertungsprogrammen zu berücksichtigen.

6.3.3.2.2 Format: „POS“-Liste

Die Datendatei ist ebenfalls im Textformat erstellt. Jede Transaktion wird jedoch in mehreren Zeilen dargestellt wobei in einer Zeile nur eine ProduktID steht. D.h. für jedes Produkt in der Transaktion wird eine eigene Zeile erstellt.

Die Zeilen sind folgendermaßen aufgebaut:

<TransaktionsID>;<Zeitpunkt>;<KundenID>;< ProduktID>

TransaktionsID: Jede Transaktion erhält eine fortlaufende, ganzzahlige ID. Anhand dieser Nummer kann erkannt werden, welche Zeilen zu einer Transaktion gehören.

Zeitpunkt: Der Zeitpunkt der Transaktion als double Zahl im Bereich [0;1]

KundenID: Ganzzahlige ID des Kunden, der die Transaktion durchführt.

ProduktID: Hier wird die ID des Produktes angeführt.

Alle Werte sind auch in diesem Format durch Strichpunkt (;) getrennt, um die Daten in anderen Programmen auswerten zu können.

Beispiel für einen Teil einer Ausgabedatei im „POS“-Format:

```
00080;0.341463;00006;005
00081;0.395607;00006;007
00082;0.422575;00006;008
00083;0.513555;00006;009
00083;0.513555;00006;010
00084;0.564935;00006;007
00085;0.616585;00006;003
00085;0.616585;00006;006
00086;0.647118;00006;007
00086;0.647118;00006;008
00087;0.698790;00006;001
00088;0.709867;00006;008
00089;0.732690;00006;006
00090;0.749956;00006;003
```

6.3.4 Ansätze für Programmerweiterungen

Um Erweiterungen in das Programm einbauen zu können, werden hier die Stellen angeführt, an denen Änderungen vorzunehmen sind um Ausgabeformate zu ändern beziehungsweise um die Berechnung der Transaktionen zu verändern.

6.3.4.1 Änderung Ausgabeformat

In der Klasse cFileSupport sind zwei Ausgabefunktionen definiert, die ein Transaktionsobjekt in die Datei schreiben. Die Methode Write(cTransaction* Transaction) wird aufgerufen, wenn als Ausgabe-Format Transaktionsliste beim Programmstart gewählt wurde. In dieser Methode kann somit das Ausgabeformat am einfachsten angepasst werden.

Die Methode WriteList(cTransaction* Transaction) wird bei dem zweite Ausgabeformat (Pos-Liste) verwendet. In dieser Funktion wird die übergebene Transaction in mehreren Zeilen in die Ausgabe Datei geschrieben.

6.3.4.2 Änderung Produktauswahl des Kunden

In der cCustomer Klasse wird in der Mehtode createTimeline() für einen Kunden festgelegt, in welchen Abständen er ein Produkt einkauft. Diese Abstände sind gemäß einem Poisson Prozess exponential verteilt. Der Parameter für die Verteilung wird beim erzeugen des Produktes ebendort abgelegt. Durch eine Änderung dieser Funktion können für einen Kunden nach einem anderen statistischen Modell die Zeitpunkte für einen Produktkauf bestimmt werden.

6.3.4.3 Änderung der Bildung einer Transaktion

Derzeit wurde in der cCustomer Klasse in der Funktion TransactionTimespan eine Mehtode implementiert um aus den einzelnen Kaufzeitpunkten der Produkte eine Transaktion zu bilden. Möchte man aus der Liste der einzelen Kaufzeitpunkt nach einer anderen Regel als der Zeitspannenregel eine Transaktion bilden, so sollte die Funktion abgeändert werden. Hier wäre z.B zu implementieren, dass eine Transaktion nie mehr als eine bestimmte Anzahl an Produkten beinhalten soll.

6.3.4.4 Unterschiedliche Kundenverhalten

Möchte man unterschiedliche Kunden programmieren, so würde es sich empfehlen Ableitungen der cCustomer Klasse zu bilden. In der Methode Run des Generators werden Instanzen der Kunden gebildet. Derzeit wird immer nur von cCustomer eine Instanz gebildet. Jedoch könnte man in der dort ablaufenden Schleife nach einem festgelegten Algorithmus unterschiedliche Kunden erzeugen, und diese die Einkäufe durchführen lassen.

7 Auswertung der erzeugten Testdaten

Für die Auswertungen wurde mit dem Testdatengenerator ein Datenbestand gebildet. Es wurden Daten für 50.000 Kunden und 30 Produkte erzeugt. Der Mean-Parameter wurde im Bereich 0 – 3 angegeben. Bei diesem Testlauf wurden 5.951.999 Zufallszahlen berechnet und 1.951.969 Transaktionszeilen gebildet. Das ergibt in diesem Datenfile 1.395.412 Transaktionen.

Als Computer wurde ein Pentium III mit 850 MHz und 256 MB Hauptspeicher unter dem Betriebsystem Windows XP verwendet. Das Datenfile war in weniger als 15 Minuten berechnet und belegt auf der Festplatte ca. 55 MB.

Das Ausgabeformat wurde auf POS-Liste gesetzt. Für eine weitere Auswertung mussten aus der Datendatei die notwendigen Informationen berechnet werden. Dies wurde mittels eines SQL-Servers bewerkstelligt. Die Daten wurden in den SQL-Server importiert.

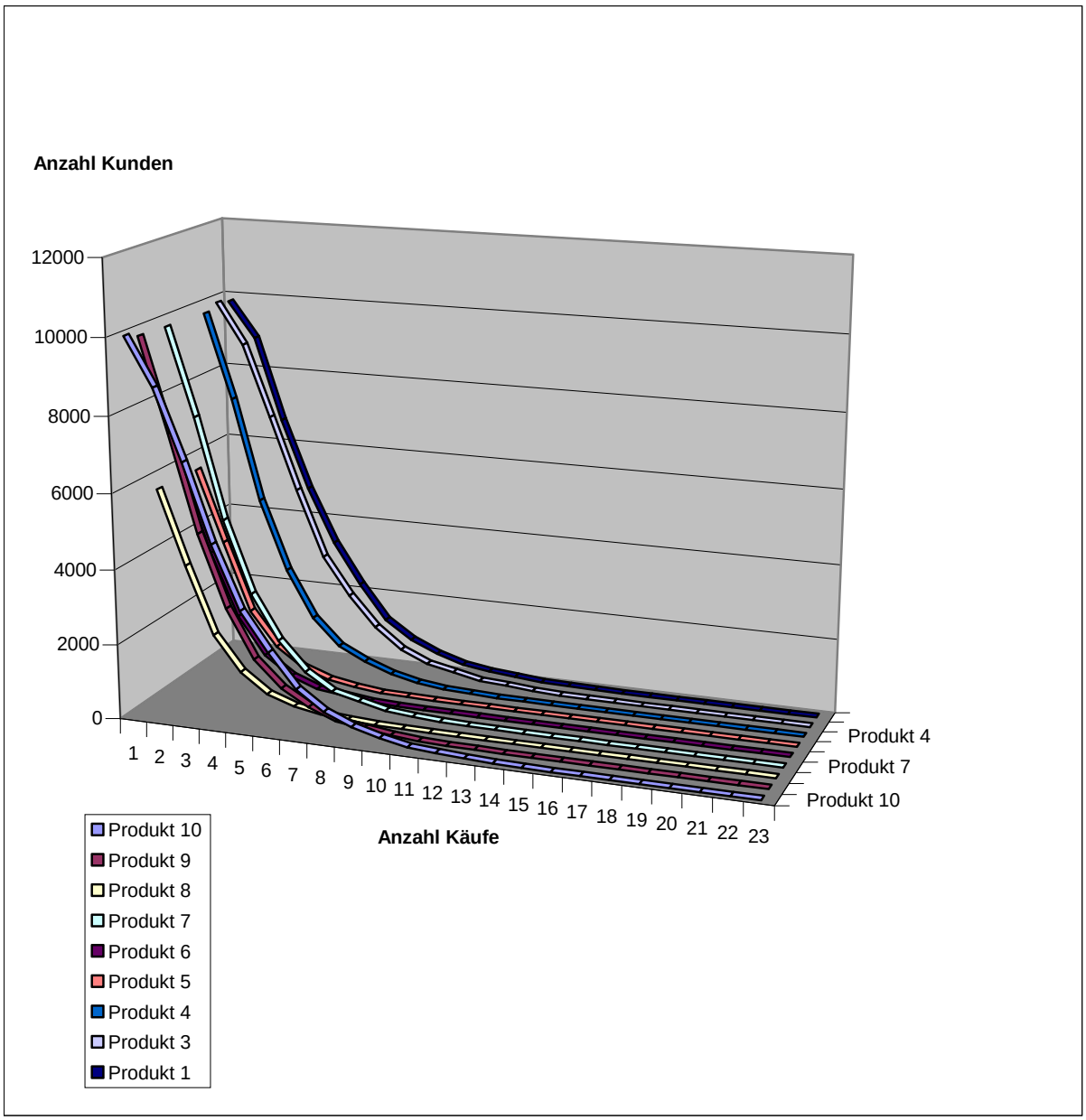
Ziel der Auswertungen ist es, eine Liste zu erhalten, die angibt, von wie vielen Kunden ein Produkt x-mal gekauft wurde. Diese Liste erhält man mit folgender SQL-Abfrage aus dem Datenbestand:

```
select anz, count(*) as nr FROM
(
     select customer, count(*) as anz from transactions
     where product LIKE '001%'
     group by customer
) as T
GROUP BY anz
order by  anz
```

Die Felder werden aus der Textdatei in die Datenbank als Textfelder übernommen. Daher wird in der Abfrage die Product-ID als Text-Parameter verwendet. Die Berechnung der Liste dauert jedoch weniger als 20 Sekunden. Daher war eine weitere Optimierung nicht notwendig.

Einen ersten Überblick der gebildeten Daten liefert folgendes Diagramm. Es wurden dabei die ersten 10 Produkte der Datendatei mittels der Abfrage berechnet und in dem Diagramm dargestellt. Das Produkt 2 wurde mit einem Mean von 0 berechnet,

und scheint somit nicht in dem Datenbestand auf. Daher sollte man die Angabe eines Mean-Wertes von 0 vermeiden.

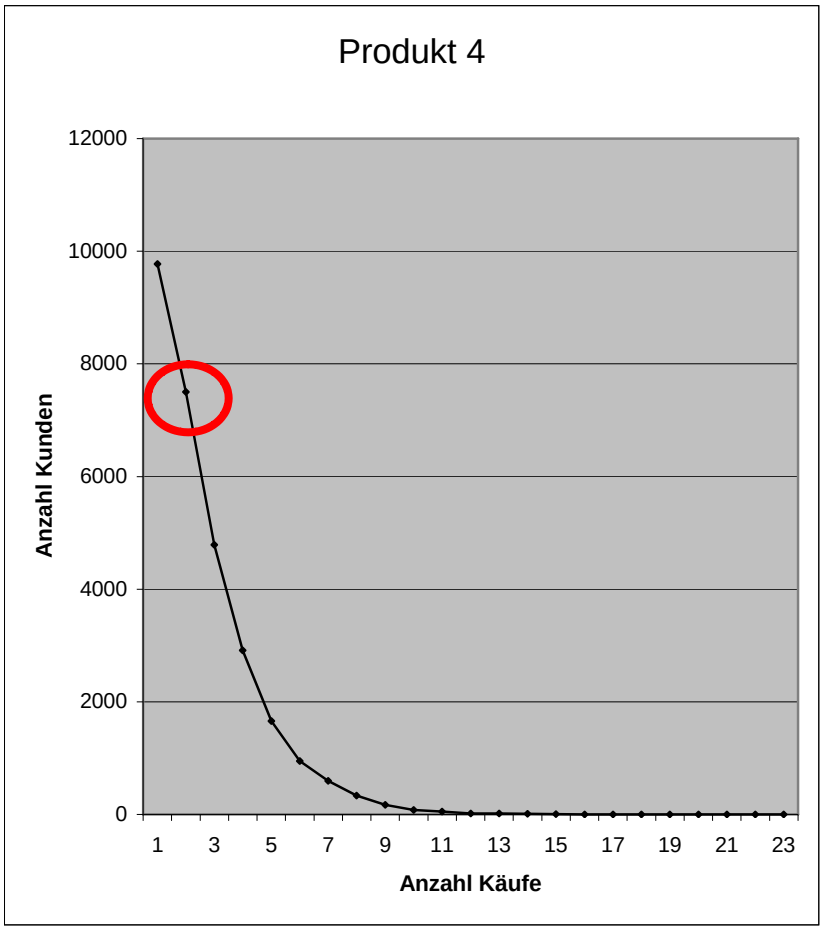


Anhand des Diagramms lässt sich die LSD-Verteilung der Kaufhäufigkeiten der Produkte erkennen.

Im Anschluss werden einige Produkte herausgegriffen und die Bedeutung des Mean-Parameters dargestellt. Dieser Parameter beschreibt wie oben bereits dargelegt, die Gamma-Verteilung der Kaufhäufigkeiten eines Kunden.

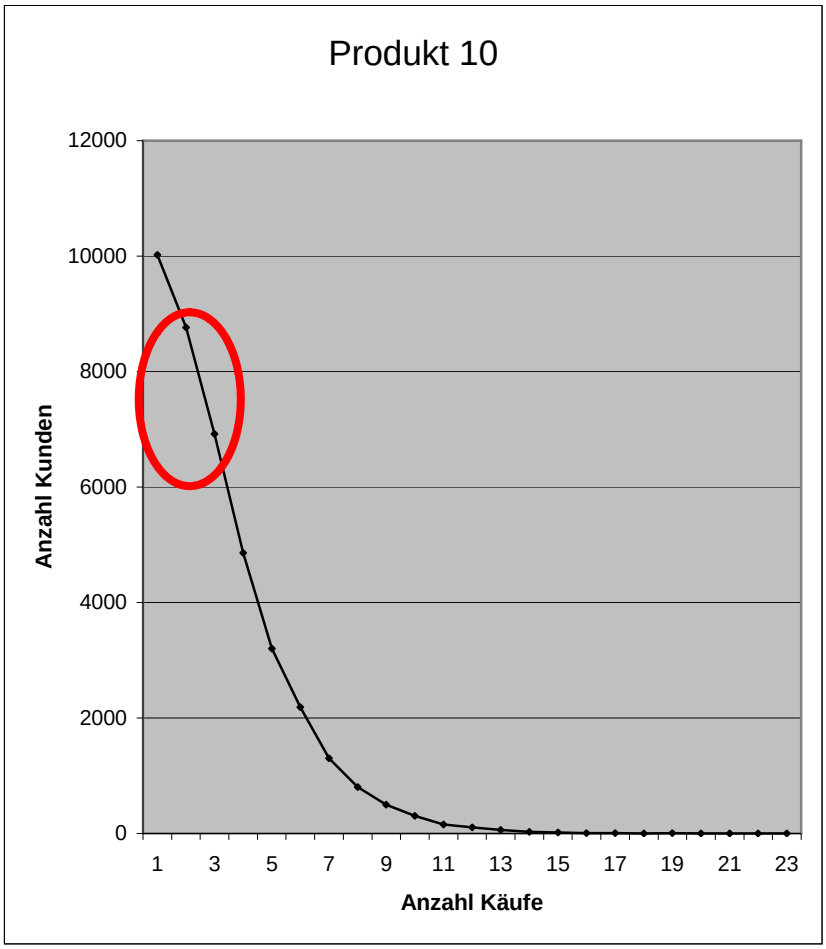
7.1 Detailbetrachtung einiger Produkte

Für das Produkt 4 wurde ein Mean von 2 für die Berechnung verwendet. So erklärt sich die hohe Anzahl an Kunden, die das Produkt zweimal gekauft haben.



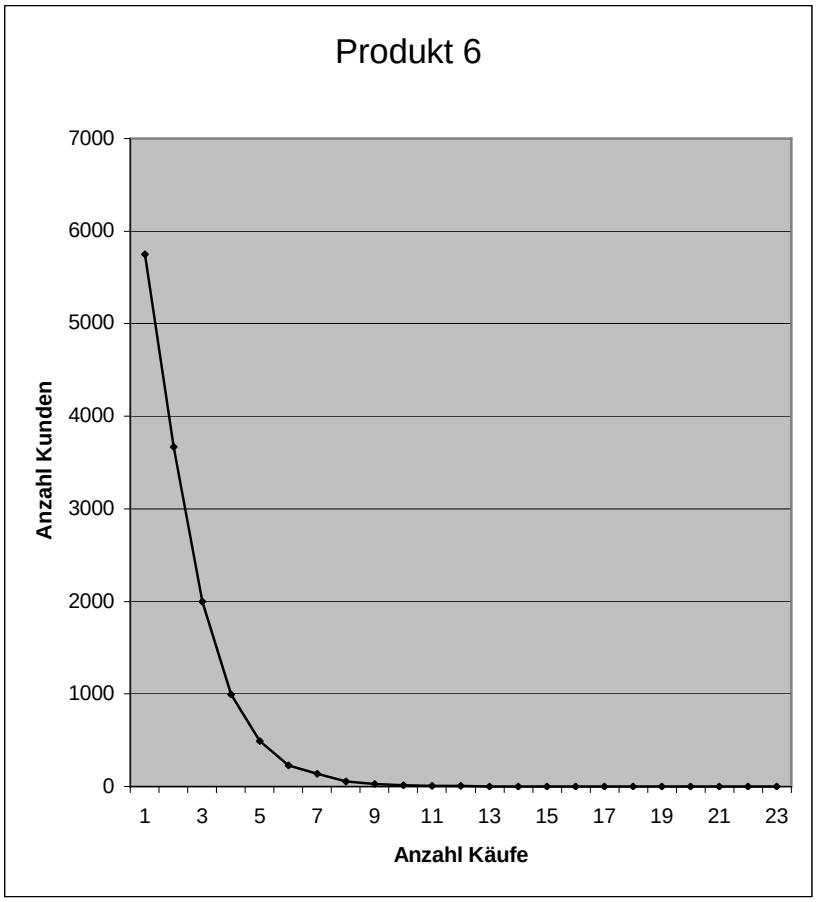
purchase freqeuncy	Produkt 4
1	9771
2	7504
3	4786
4	2916
5	1658
6	950
7	599
8	333
9	172
10	81
11	52
12	17
13	19
14	11
15	6
16	2
17	2
18	0
19	1
20	0
21	0
22	1
23	0

Die nächste Abbildung zeigt das Produkt 10, das eine Mean von 3 hat. Daher erkennt man im Diagramm den Knick bei den Kunden die zweimal oder dreimal das Produkt gekauft haben.



purchase freqeuncy	Produkt 10
1	10023
2	8764
3	6920
4	4859
5	3200
6	2191
7	1303
8	802
9	498
10	303
11	156
12	104
13	62
14	30
15	16
16	3
17	7
18	2
19	4
20	0
21	0
22	0
23	1

Als letztes betrachten wir das Produkt 6 mit einem Mean von 1. Dieses Diagramm zeigt eine typische LSD-Verteilung. Die Kurve fällt sofort nach den Kunden die das Produkt einmal gekauft haben stark ab.



purchase freqeuncy	Produkt 6
1	5751
2	3672
3	1994
4	995
5	493
6	228
7	139
8	55
9	25
10	12
11	9
12	6
13	3
14	3
15	1
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0

Aus den angeführten Diagrammen mit einem Mean von 2, 3 oder 1 erkennt man, dass die Angabe des Mean-Parameters ein wichtiger Faktor bei der Bildung der Daten ist. Für die Analyse der Daten ist es daher wichtig, festzustellen mit welchem Mean-Wert ein bestimmtes Produkt berechnet wurde. Diese Informationen können aus dem produzierten HTML-File entnommen werden.

8 Sourcecode

8.1 Datei: TDG.h

```
#include <map>
#include <string>
#include <vector>
#include <math.h>
#include <assert.h>
#include <stdio.h>

using namespace std ;
#define StringType string
// tell the compiler to shut up
#pragma warning(disable:4786)


/*-------------------------------------------------------------------------

Dieser Teil ist für die Auswertung der Übergabeparameter


-------------------------------------------------------------------------*/
struct CCmdParam
{
   vector<StringType> m_strings;
};
// this class is actually a map of strings to vectors
typedef map<StringType, CCmdParam> _CCmdLine;

// the command line parser class
class CCmdLine : public _CCmdLine
{
public:
   int         SplitLine(int argc, char **argv);
   bool        HasSwitch(const char *pSwitch);
   StringType  GetSafeArgument(const char *pSwitch, int iIdx, const char
*pDefault);
   StringType  GetArgument(const char *pSwitch, int iIdx);
   int         GetArgumentCount(const char *pSwitch);

protected:
   bool        IsSwitch(const char *pParam);
};
/*-------------------------------------------------------------------------
----

Klassen für die Zufallszahlberechnung



-------------------------------------------------------------------------
--*/
#ifdef HIGH_RESOLUTION
typedef long double trfloat;        // use long double precision
#else
typedef double trfloat;             // use double precision
#endif


class TRandomMersenne {                 // encapsulate random number generator
  enum constants {
#if 1
    // define constants for MT11213A
```

```
    N = 351, M = 175, R = 19, MATRIX_A = 0xEABD75F5,
    TEMU = 11, TEMS = 7, TEMT = 15, TEML = 17, TEMB = 0x655E5280, TEMC = 0xFFD58000
#else
    // or constants for MT19937
    N = 624, M = 397, R = 31, MATRIX_A = 0x9908B0DF,
    TEMU = 11, TEMS = 7, TEMT = 15, TEML = 18, TEMB = 0x9D2C5680, TEMC = 0xEFC60000
#endif
  };
  public:
  TRandomMersenne(long int seed) {           // constructor
    RandomInit(seed);}
  void RandomInit(long int seed);            // re-seed
  long IRandom(long min, long max);          // output random integer
  double Random();                           // output random float
  unsigned long BRandom();                   // output random bits
  long count;

private:
  unsigned long mt[N];                       // state vector
  int mti;                                   // index into mt

  };

/*-----------------------------------------------------------------------------

 Classes for Generator:

-----------------------------------------------------------------------------*/
class cMyRandom
{
private:
       TRandomMersenne* mRND;

public:
       cMyRandom() ;

       ~cMyRandom(void);
       double GetExponential(double mean);
       double GetUniform();
       int GetUniform(long min, long max);
       double GetGamma(double a);
       long GetCount();
};



class cListItem
{
private:
       cListItem* mNext;
       double mKey;

public:
       cListItem(void);
       ~cListItem(void);
       cListItem* GetNext(void);
       void SetNext(cListItem* pItem);
       double GetKey();
       void SetKey(double Key);

       virtual void Dump();
       virtual void Dispose();

};


class cList
{
```

```
private:
	cListItem First;
	cListItem Last;
	cListItem* curItem;
	cListItem* lastItem;

public:
	cList(void);
	~cList(void);
	void Add(cListItem* pItem);
	void Insert(cListItem* pItem);
	void Dump();
	cListItem* Read();
	long Count;
	void MoveFirst(void);
};

class cProduct : public cListItem
{
private:
	long mGamma; // used for gamma distribution of Customers of this product
	cMyRandom* pTheRandom;

public:
	long ProductID;
	cProduct(cMyRandom* theRandom, long Mean);
	cProduct(cMyRandom* theRandom,int ID, long Mean);
	~cProduct(void);
	void Dump();
	double GetMean();
	long GetGamma();
};

class cCustomer
{
private:
	cList* pProductList;
	cList* pShoppingList;
	cMyRandom* pTheRandom;
	void goShopping(void);
	void CreateTimeLine(double Gamma, cProduct* pProduct);

public:
	long ID;
	cCustomer(cList* ProductList, cMyRandom* Random,long id);
	~cCustomer(void);
	void Dump();
	void TransactionTimespan(cList* TransactionList, double TimeSpan);
};

class cTransaction : public cListItem
{
private:
	double Time;

public:
	cList* pProductList;
	long CustomerID;
	cTransaction(double Time, long theCustomerID);
	~cTransaction(void);
	void AddProduct(long ProductID);
	void Dump(void);
	void Dispose(void);
};

class cShoppingDate : public cListItem
{
public:
```

```
	cProduct* pProduct;
	cShoppingDate(double Time, cProduct* Product);
	~cShoppingDate(void);
	void Dump(void);
};


class cFileSupport
{
public:
	FILE *stream;
	cFileSupport(void);
	cFileSupport(char* fName);
	~cFileSupport(void);

	void Close(void);
	int Write(double db);
	int Write(cTransaction* Transaction);
	int Write(char* c);
	int Write(int i);
	int Write(long i);
	int WriteList(cTransaction* Transaction);
};

class cHTMLReport
{
public:
	cHTMLReport(void);
	cHTMLReport(char* fName);
	~cHTMLReport(void);

	long CountFigures;
	long CountCustomers;
	long CountProducts;
	long MeanMin;
	long MeanMax;
	double TimeSpan;

	void CreateFile(cList* pProducts);


private:
	cFileSupport *FS;

};

class cGenerator
{
private:
	cList* pTheTransactionList;
	cList* pTheProductList;
	int mCountProducts;
	int mMeanMin;
	int mMeanMax;
	double mTimeSpan;
	bool mPOSOutput;

public:
	cMyRandom* pTheRandom;
	cGenerator(void);
	~cGenerator(void);
	void Run(int ProductCount,int CustomerCount,int MeanMin, int MeanMax,const
char* Filename, double TimeSpan, bool POSOutput);
	void CreateProducts(void);
};
```

8.2 Datei: TDG.cpp

```
#include <time.h>               // define time()
#define _USE_MATH_DEFINES // für M_PI!
#include <math.h>

#include "TDG.h"


int CCmdLine::SplitLine(int argc, char **argv)
{
   clear();

   StringType curParam; // current argv[x]

   // skip the exe name (start with i = 1)
   for (int i = 1; i < argc; i++)
   {
      // if it's a switch, start a new CCmdLine
      if (IsSwitch(argv[i]))
      {
         curParam = argv[i];
         StringType arg;
         // look at next input string to see if it's a switch or an argument
         if (i + 1 < argc)
         {
            if (!IsSwitch(argv[i + 1]))
            {
               // it's an argument, not a switch
               arg = argv[i + 1];

               // skip to next
               i++;
            }
            else
            {
               arg = "";
            }
         }

         // add it
         CCmdParam cmd;

         // only add non-empty args
         if (arg != "")
         {
            cmd.m_strings.push_back(arg);
         }

         // add the CCmdParam to 'this'
         pair<CCmdLine::iterator, bool> res = insert(CCmdLine::value_type(curParam,
cmd));

      }
      else
      {
         // it's not a new switch, so it must be more stuff for the last switch

         // ...let's add it
             CCmdLine::iterator theIterator;

         // get an iterator for the current param
         theIterator = find(curParam);
             if (theIterator!=end())
         {
            (*theIterator).second.m_strings.push_back(argv[i]);
         }
```

```
         else
         {
            // ??
         }
      }
   }

   return (int)size();
}

/*------------------------------------------------------

   protected member function
   test a parameter to see if it's a switch :

   switches are of the form : -x
   where 'x' is one or more characters.
   the first character of a switch must be non-numeric!

------------------------------------------------------*/

bool CCmdLine::IsSwitch(const char *pParam)
{
   if (pParam==NULL)
      return false;

   // switches must non-empty
   // must have at least one character after the '-'
   int len = (int)strlen(pParam);
   if (len <= 1)
   {
      return false;
   }

   // switches always start with '-'
   if (pParam[0]=='-')
   {
      // allow negative numbers as arguments.
      // ie., don't count them as switches
      return (!isdigit(pParam[1]));
   }
   else
   {
      return false;
   }
}

/*------------------------------------------------------
   bool CCmdLine::HasSwitch(const char *pSwitch)

   was the switch found on the command line ?

   ex. if the command line is : app.exe -a p1 p2 p3 -b p4 -c -d p5

   call                        return
   ----                        ------
   cmdLine.HasSwitch("-a")     true
   cmdLine.HasSwitch("-z")     false
------------------------------------------------------*/

bool CCmdLine::HasSwitch(const char *pSwitch)
{
      CCmdLine::iterator theIterator;
      theIterator = find(pSwitch);
      return (theIterator!=end());
}

/*------------------------------------------------------
```

```
   StringType CCmdLine::GetSafeArgument(const char *pSwitch, int iIdx, const char
*pDefault)

   fetch an argument associated with a switch . if the parameter at
   index iIdx is not found, this will return the default that you
   provide.

   example :

   command line is : app.exe -a p1 p2 p3 -b p4 -c -d p5

   call                                    return
   ----                                    ------
   cmdLine.GetSafeArgument("-a", 0, "zz")    p1
   cmdLine.GetSafeArgument("-a", 1, "zz")    p2
   cmdLine.GetSafeArgument("-b", 0, "zz")    p4
   cmdLine.GetSafeArgument("-b", 1, "zz")    zz

-------------------------------------------------------*/

StringType CCmdLine::GetSafeArgument(const char *pSwitch, int iIdx, const char
*pDefault)
{
   StringType sRet;

   if (pDefault!=NULL)
      sRet = pDefault;

   try
   {
      sRet = GetArgument(pSwitch, iIdx);
   }
   catch (...)
   {
   }

   return sRet;
}

/*-------------------------------------------------------

   StringType CCmdLine::GetArgument(const char *pSwitch, int iIdx)

   fetch a argument associated with a switch. throws an exception
   of (int)0, if the parameter at index iIdx is not found.

   example :

   command line is : app.exe -a p1 p2 p3 -b p4 -c -d p5

   call                            return
   ----                            ------
   cmdLine.GetArgument("-a", 0)     p1
   cmdLine.GetArgument("-b", 1)     throws (int)0, returns an empty string

-------------------------------------------------------*/

StringType CCmdLine::GetArgument(const char *pSwitch, int iIdx)
{
   if (HasSwitch(pSwitch))
   {
         CCmdLine::iterator theIterator;

      theIterator = find(pSwitch);
         if (theIterator!=end())
      {
         if ((int)(*theIterator).second.m_strings.size() > iIdx)
```

```
         {
            return (*theIterator).second.m_strings[iIdx];
         }
      }
   }

   throw (int)0;

   return "";
}

/*-------------------------------------------------------
   int CCmdLine::GetArgumentCount(const char *pSwitch)

   returns the number of arguments found for a given switch.

   returns -1 if the switch was not found

-------------------------------------------------------*/

int CCmdLine::GetArgumentCount(const char *pSwitch)
{
   int iArgumentCount = -1;

   if (HasSwitch(pSwitch))
   {
          CCmdLine::iterator theIterator;

      theIterator = find(pSwitch);
          if (theIterator!=end())
      {
         iArgumentCount = (int)(*theIterator).second.m_strings.size();
      }
   }

   return iArgumentCount;
}


void TRandomMersenne::RandomInit(long int seed) {
  // re-seed generator
       count = 0;

  unsigned long s = (unsigned long)seed;
  for (mti=0; mti<N; mti++) {
    s = s * 29943829 - 1;
    mt[mti] = s;}}


unsigned long TRandomMersenne::BRandom() {
  // generate 32 random bits
  count++;
       unsigned long y;

  if (mti >= N) {
    // generate N words at one time
    const unsigned long LOWER_MASK = (1u << R) - 1; // lower R bits
    const unsigned long UPPER_MASK = -1 << R;       // upper 32-R bits
    int kk, km;
    for (kk=0, km=M; kk < N-1; kk++) {
      y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK);
      mt[kk] = mt[km] ^ (y >> 1) ^ (-(signed long)(y & 1) & MATRIX_A);
      if (++km >= N) km = 0;}

    y = (mt[N-1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ (-(signed long)(y & 1) & MATRIX_A);
    mti = 0;}
```

```
  y = mt[mti++];

  // Tempering (May be omitted):
  y ^=  y >> TEMU;
  y ^= (y << TEMS) & TEMB;
  y ^= (y << TEMT) & TEMC;
  y ^=  y >> TEML;

  return y;}


double TRandomMersenne::Random() {
  // output random float number in the interval 0 <= x < 1
  union {
    double f;
    unsigned long i[2];}
  convert;
  // get 32 random bits and convert to float
  unsigned long r = BRandom();
  convert.i[0] =  r << 20;
  convert.i[1] = (r >> 12) | 0x3FF00000;
  return convert.f - 1.0;}


long TRandomMersenne::IRandom(long min, long max) {
  // output random integer in the interval min <= x <= max
  long r;
  r = long((max - min + 1) * Random()) + min; // multiply interval with random and
truncate
  if (r > max) r = max;
  if (max < min) return 0x80000000;
  return r;}



cMyRandom::cMyRandom()
{
      mRND =  new TRandomMersenne((long)time(0));
}

cMyRandom::~cMyRandom(void)
{
      delete mRND;
}
double cMyRandom::GetExponential(double mean)
{
      double x;
      x = mRND->Random();

      double t;
      t = (-log(1-x))/(mean);

      return t;
}
double cMyRandom::GetUniform()
{
      return mRND->Random();
}
int cMyRandom::GetUniform(long min, long max)
{
      int x = mRND->IRandom(min,max);
      return x;
}
double cMyRandom::GetGamma(double a)
{     double x=0.0;

      if(a<6)
      {
```

```
		x=1.0;
		for(int j=1;j<=a;j++)
		{
			double u = this->GetUniform();
			x *= u;
		}
		x= -log(x);
	}
	else
	{

		double U = 0.0;
		double Y = 0.0;
		bool OK = false;

		while (!OK)
		{	//A1
			U = this->GetUniform();

			Y = tan(M_PI * U);
			x = sqrt(2* a -1)* Y + a -1;
			if (x <= 0)
				OK=false;// back to A1
			else
				OK =true;

			if (OK)
			{// A2
				double V = this->GetUniform();
				double temp=0.0;
				temp = (1+Y*Y) * exp((a-1)*log(x/(a-1))-sqrt(2*a-1)*Y);
				if (V>temp)
					OK = false; // back to  A1
			}
		}
	}
	return(x);
}

long cMyRandom::GetCount()
{
	return mRND->count;
}



cFileSupport::cFileSupport(void)
{
	stream = fopen("output.txt","w");
}
cFileSupport::cFileSupport(char* fName)
{
	stream = fopen(fName,"w");
}



cFileSupport::~cFileSupport(void)
{
	fclose(stream);

}

int cFileSupport::Write(char* txt)
{
	fprintf (stream, txt);
```

```
	fprintf (stream,"\r\n");


	return 0;
}
int cFileSupport::Write(double db)
{
	char txt[57];
	sprintf(txt,"%f",db);
	if (txt[0] != '-')
		txt[1] = ',';
	else
		txt[2] = ',';

	this->Write(txt);

	return 0;

}
int cFileSupport::Write(int i)
{
	char txt[57];
	sprintf(txt,"%i",i);
	this->Write(txt);
	return 0;
}
int cFileSupport::Write(long i)
{
	char txt[57];
	sprintf(txt,"%i",i);
	this->Write(txt);
	return 0;
}

int cFileSupport::Write(cTransaction* Transaction)
{
	static long TranID=0;

	char txt[200];
	char buf[200];
	// TransactionID
	sprintf(buf,"%5.5d;",TranID++);
	strcpy(txt,buf);

	// TimeStamp
	sprintf(buf,"%f;",Transaction->GetKey());
	strcat(txt,buf);

	//CustomerID
	sprintf(buf,"%5.5d;",Transaction->CustomerID);
	strcat(txt,buf);

	cProduct* pProd;
	Transaction->pProductList->MoveFirst();
	while ( (pProd=(cProduct*)Transaction->pProductList->Read())!=NULL)
	{
		sprintf(buf,"%3.3d;",(int)pProd->GetKey() );
		strcat(txt,buf);
	}
	this->Write(txt);
	return 0;
}
int cFileSupport::WriteList(cTransaction* Transaction)
{
static long TranID=0;

	char txt[200];
	char buf[200];
```

```
	// TransactionID
	sprintf(buf,"%5.5d;",TranID++);
	strcpy(txt,buf);

	// TimeStamp
	sprintf(buf,"%f;",Transaction->GetKey());
	strcat(txt,buf);

	//CustomerID
	sprintf(buf,"%5.5d;",Transaction->CustomerID);
	strcat(txt,buf);

	cProduct* pProd;
	Transaction->pProductList->MoveFirst();
	while ( (pProd=(cProduct*)Transaction->pProductList->Read())!=NULL)
	{
		sprintf(buf,"%s%3.3d",txt,(int)pProd->GetKey() );
		this->Write(buf);
	}

	return 0;
}
void cFileSupport::Close(void)
{
	fclose(stream);
}


cGenerator::cGenerator(void)
{
	mMeanMin = 1;
	mMeanMax = 3;
	pTheRandom = new cMyRandom();

	pTheProductList = new cList();
	mTimeSpan = 0.01;
}

cGenerator::~cGenerator(void)
{
	delete pTheRandom;

	delete pTheProductList;
}

void cGenerator::Run(int ProductCount, int CustomerCount, int MeanMin, int MeanMax,
const char* Filename, double TimeSpan, bool POSOutput)

{
	mTimeSpan = TimeSpan;
	mMeanMin = MeanMin;
	mMeanMax = MeanMax;

	mPOSOutput = POSOutput;

	mCountProducts = ProductCount;
	CreateProducts();



	// Open File for Output!
	char oDatei[500];
	strcpy(oDatei, Filename);
	strcat(oDatei,".txt");
	long tCount = 0;

	cFileSupport* F = new cFileSupport(oDatei);
```

```
	for (int i = 0;i<CustomerCount;i++)
	{
		cCustomer* pCustomer = new cCustomer(pTheProductList,pTheRandom,i);
		// create a new List

		cList* myTList = new cList();

		pCustomer->TransactionTimespan(myTList,TimeSpan);
		delete pCustomer;

		cTransaction* pTran;
		myTList->MoveFirst();
		while ((pTran=(cTransaction*)myTList->Read())!=NULL)
		{
			// Which kind of Output format is requested?
			if (mPOSOutput)
				F->WriteList(pTran);
			else
				F->Write(pTran);
		}

		tCount += myTList->Count;

		printf("Customer:%i  TranCount:%i\n", i,tCount);
		delete myTList;
	}



	// Great work! Close the file!
	F->Close();
	printf ("Number of Figures: %i", pTheRandom->GetCount());

	char hDatei[500];
	strcpy(hDatei, Filename);
	strcat(hDatei,".html");

	cHTMLReport*  rep = new cHTMLReport(hDatei);
	rep->CountCustomers = CustomerCount;
	rep->CountProducts = ProductCount;
	rep->CountFigures = pTheRandom->GetCount();
	rep->MeanMin = mMeanMin;
	rep->MeanMax = mMeanMax;
	rep->TimeSpan = mTimeSpan;

	rep->CreateFile(pTheProductList);

	delete rep;
	delete F;

}

void cGenerator::CreateProducts(void)
{
	long m;
	for (int i= 0;i<mCountProducts;i++)
	{
		m = pTheRandom->GetUniform(mMeanMin, mMeanMax);
		pTheProductList->Insert(new cProduct(pTheRandom,i,m));
	}
}


cShoppingDate::cShoppingDate(double Time, cProduct* Product)
{
	this->SetKey(Time);
	this->pProduct = Product;
```

```
}

cShoppingDate::~cShoppingDate(void)
{
}
void  cShoppingDate::Dump()
{
	long ID;
	ID = pProduct->ProductID;

	printf("cShoppingDate::Dump %f  Product: %d\n", this->GetKey(), this-
>pProduct->ProductID );
}


cCustomer::cCustomer(cList* ProductList, cMyRandom* Random, long id)
{
	this->ID = id;
	this->pProductList=ProductList;
	this->pShoppingList = new cList();
	this->pTheRandom = Random;
	goShopping();
}

cCustomer::~cCustomer(void)
{
	delete pShoppingList;
}


void cCustomer::goShopping(void)
{
	cProduct* Prod;
	pProductList->MoveFirst();
	while ((Prod=(cProduct*)pProductList->Read())!=NULL) // go through all
products in list
	{
		double Gamma = Prod->GetMean(); //this is the parameter of
Poissondistribution!

		CreateTimeLine(Gamma, Prod);
	}
}
void cCustomer::CreateTimeLine(double Gamma, cProduct* pProduct)
{
	double AktTime = 0.00;

	while ((AktTime+=pTheRandom->GetExponential((int)Gamma)) <1)
	{
		cShoppingDate* pZ = new cShoppingDate(AktTime, pProduct);
		pShoppingList->Insert(pZ);
	}
}
void cCustomer::Dump()
{
	pShoppingList->Dump();
}
void cCustomer::TransactionTimespan(cList* TransactionList, double TimeSpan)
{	/*
		add all transactions of this customer to the TransactionList
		criteria for new transaction is TimeSpan
	*/

	cShoppingDate* pZ;
	cTransaction* pTrans = NULL;
	double letzteTime = 0.00;
	pShoppingList->MoveFirst();
	while ( (pZ=(cShoppingDate*)pShoppingList->Read())!=NULL)
```

```
	{
		if ((letzteTime == 0.00) || (letzteTime + TimeSpan < pZ->GetKey()))
		{	// need to create a new transaction
			if (pTrans != NULL)
				TransactionList->Insert(pTrans);
			letzteTime = pZ->GetKey();
			pTrans = new cTransaction(letzteTime, this->ID );
		}
		// Add Product to transaction!
		pTrans->AddProduct((long)pZ->pProduct->GetKey());
		/*
		Set time of transaction to time of last buy
		*/
		pTrans->SetKey(pZ->GetKey());
	}
	if (pTrans != NULL)
		TransactionList->Add(pTrans);

}

cList::cList(void)
: Count(0)
{
	First.SetNext(&Last);
	curItem = NULL;
	lastItem = NULL;
}

cList::~cList(void)
{
	cListItem* it = First.GetNext();

	while (it != &Last && it->GetNext()!=&Last )
	{
		cListItem* temp;
		temp = it->GetNext();
		it->Dispose();
		delete it;
		it = temp;
	}
}
void cList::Add(cListItem* pItem)
{
	pItem->SetNext(lastItem->GetNext());
	lastItem->SetNext(pItem);
	lastItem = pItem;
	Count++;
}
void cList::Insert(cListItem* pItem)
{

	cListItem* it = &First;

	if (lastItem != NULL)
		if (lastItem->GetKey() <= pItem->GetKey())
			it =lastItem;

	while (it->GetNext()!=&Last && it->GetNext()->GetKey() < pItem->GetKey() )
		it=it->GetNext();

	pItem->SetNext(it->GetNext());
	it->SetNext(pItem);
	lastItem = pItem;
	Count++;
}


void cList::Dump()
```

```
{
	cListItem* it = &First;
	while (it->GetNext() != &Last)
	{
		it = it->GetNext();
		it->Dump();
	}
}
cListItem* cList::Read()
{
	if (curItem==NULL)
		curItem = &First;

	while (curItem->GetNext() != &Last)
	{
		curItem = curItem->GetNext();
		return curItem;
	}
	return NULL;
}
void cList::MoveFirst(void)
{
	curItem = NULL;
}


cListItem::cListItem(void)
{
	mNext = NULL;
}

cListItem::~cListItem(void)
{

}

cListItem* cListItem::GetNext()
{
	return mNext;
}
void cListItem::SetNext(cListItem* pItem)
{
	if (pItem == this)
		printf("Problem");


	mNext = pItem;
}
double cListItem::GetKey()
{
	return mKey;
}
void cListItem::SetKey(double Key)
{
	mKey = Key;
}
void cListItem::Dump()
{
	printf("cListItem::Dump %f \n",this->GetKey());
}
void cListItem::Dispose()
{
	// use this function in child classes to
	// free memory!
}

cProduct::cProduct(cMyRandom* TheRandom, long Mean)
{
```

```
	pTheRandom = TheRandom;
	ProductID = 0;
	this->SetKey((double)ProductID);
	mGamma = Mean;

}
cProduct::cProduct(cMyRandom* TheRandom,int ID, long Mean)
{
	pTheRandom = TheRandom;
	ProductID = ID;
	this->SetKey((double)ID);
	mGamma = Mean;
}
cProduct::~cProduct(void)
{
}
void cProduct::Dump()
{
	printf("cProduct::Dump %f \n", this->GetKey());
}
double cProduct::GetMean()
{
	return pTheRandom->GetGamma(mGamma);
}
long cProduct::GetGamma()
{
	return mGamma;
}

cTransaction::cTransaction(double Time, long theCustomerID)
{
	this->CustomerID = theCustomerID;
	this->Time = Time;
	this->pProductList = new cList();
	this->SetKey(Time);
}

cTransaction::~cTransaction(void)
{

}

void cTransaction::AddProduct(long ProductID)
{
	cListItem* pItem = new cListItem();
	pItem->SetKey((double)ProductID);
	pProductList->Insert(pItem);
}

void cTransaction::Dump(void)
{
	printf("**** %f - %i\n",this->Time, CustomerID );
	pProductList->Dump();
}
void cTransaction::Dispose(void)
{
	delete pProductList;
}



void showHelp()
{
	printf("TestDatenGenerator TDG\n");
	printf("----------------------\n");
	printf("Parameter:\n");
	printf("-h   Helpscreen (you are reading the help)\n");
	printf("-c   Number of Customer\n");
```

```
	printf("-p   Number of Products\n");
	printf("-m   x y Mean of Products (between x and y\n\n");
	printf("-o   name of the outputfile\n");
	printf("-z   TimeSpan of one transaction\n");
	printf("-tlist output is grouped by transaction\n");
	printf("without -tlis: output is in POS-Format\n");

	printf("possible usage:\n");
	printf("-c 20 -p 10 -m 1 4");


}

// Main entry point:
int main(int argc, char **argv)
{
	cGenerator* pTheGenerator = new cGenerator();
	/*
	the Startup parameter have to be set here
	then we can call RUN()
	*/

	CCmdLine* cmdLine = new CCmdLine();

	cmdLine->SplitLine(argc, argv) ;


	if (cmdLine->HasSwitch("-h"))
	{
		showHelp();
		exit(0);
	}


	int Customer = atoi(cmdLine->GetSafeArgument("-c",0,"20").c_str());
	int ProductCount = atoi(cmdLine->GetSafeArgument("-p",0,"10").c_str());
    int MeanMin = atoi(cmdLine->GetSafeArgument("-m",0,"1").c_str());
	int MeanMax = atoi(cmdLine->GetSafeArgument("-m",1,"4").c_str());
	double TimeSpan = atof(cmdLine->GetSafeArgument("-t",0,"0.01").c_str());

	const char* fName;
	fName = (const char*)cmdLine->GetSafeArgument("-o",0,"output").c_str();

	bool POSOutput = true;
	// if we have the switcht -tlist we a
	// list of transaction and not a list like
	// Point of sale output
	if (cmdLine->HasSwitch("-tlist"))
		POSOutput = false;

	pTheGenerator->Run(ProductCount, Customer,MeanMin,MeanMax,fName,
TimeSpan,POSOutput);

	delete pTheGenerator;
	delete cmdLine;
	return 0;
}



cHTMLReport::cHTMLReport(char *fName)
{
	FS = new cFileSupport(fName);
}

cHTMLReport::cHTMLReport()
{
	FS = new cFileSupport("output.html");
```

```
}

cHTMLReport::~cHTMLReport()
{
	delete FS;
}

void cHTMLReport::CreateFile(cList *pProducts)
{
	FS->Write("<HTML>");
	FS->Write("<H1>Erzeugte Daten:</H1><BR>");
	FS->Write("<TABLE>");
	FS->Write("<TR>");

	FS->Write("<TD>Anzahl Customer:</TD><TD>");
	FS->Write(CountCustomers);
	FS->Write("</TD></TR>");
	FS->Write("<TR><TD>Anzahl Products:</TD><TD>");
	FS->Write(CountProducts);
	FS->Write("</TD></TR>");

	FS->Write("<TR><TD>Mean Minimum:</TD><TD>");
	FS->Write(MeanMin);
	FS->Write("</TD></TR>");

	FS->Write("<TR><TD>Mean Maximum:</TD><TD>");
	FS->Write(MeanMax);
	FS->Write("</TD></TR>");

	FS->Write("<TR><TD>Anzahl Zufallszahlen:</TD><TD>");
	FS->Write(CountFigures);
	FS->Write("</TD></tr>");

	FS->Write("<TR><TD>TimeSpan:</TD><TD>");
	FS->Write(TimeSpan);


	FS->Write("</TD></TR></TABLE><HR>");

	// the Products:
	FS->Write("<H2>Products:<H2>");
	FS->Write("<TABLE BORDER=1>");
	FS->Write("<TR><TD>ProductNR</td><TD>Mean</TD>");
	cProduct* pProd;
	pProducts->MoveFirst();
	while ((pProd=(cProduct*)pProducts->Read())!=NULL)
	{
		FS->Write("<TR><TD>");
		FS->Write(pProd->ProductID);
		FS->Write("</TD><TD>");
		FS->Write((int) pProd->GetGamma());
		FS->Write("</TD></TR>");
	}
	FS->Write("</TABLE>");

	FS->Write("</HTML>");

	FS->Close();
}
```

9 Abbildungsverzeichnis

Abb: 1 Beispiel für Taxanomien zwischen zwei Artikel 11

Abb: 2 Einkauf eines Produktes 25

Abb: 3 Zeitachse nach allen Produkten 25

Abb: 4 gebildete Transaktionen 26

Abb: 5 Exponential Verteilung ..... **Fehler! Textmarke nicht definiert.**

Abb: 6 Gamma-Verteilung ..... **Fehler! Textmarke nicht definiert.**

Abb: 7 Einfacher Kongruenzgenerator 32

Abb: 8 Großer Kongruenzgenerator 33

Abb: 9 erzeugte Exponentialverteilung 36

Abb: 10 Algorithmus für Gammaverteilung 37

Abb: 11 Erzeugte Gammaverteilung 38

Abb: 12 Klassendiagramm 41

Abb: 13 Sequenzdiagramm: Run 52

Abb: 14 Sequenzdiagramm: goShopping 54

Abb: 15 Sequenzdiagramm TransactionenTimespan 56

Abb: 16 erzeugte Daten 1 ..... **Fehler! Textmarke nicht definiert.**

Abb: 17 erzeugte Daten 2 ..... **Fehler! Textmarke nicht definiert.**

Abb: 18 erzeugte Daten 3 ..... **Fehler! Textmarke nicht definiert.**

Abb: 19 erzeugte Daten 4 ..... **Fehler! Textmarke nicht definiert.**

Abb: 20 erzeugte Daten 5 ..... **Fehler! Textmarke nicht definiert.**

10 Literaturverzeichnis

[AGRSRI94]
Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Proceedings of the 20th VLDB Conference, 1994.

[AIS93]
Rakesh Agrawal, Tomasz Imielinski, and Arun Swami,Mining association rules between sets of items in large databases. In Proceedings of the 1993 ACM SIGMOD Conference, 1993.

[Alma02]
Almaden Research Centre, http://www.almaden.ibm.com/almaden/welcome.html, Abruf vom 4.11.02

[Animal99]
CCmdLine , A utility for parsing command line, Christ Losinger 1999, http://www.smalleranimals.com , Abruf vom 3.10.2002

[Baum98]
Joachim Baumeister, Data Mining Verfahren zum Finden von Assoziationsregeln und Evaluierung durch das Werkzeug MISTRAL, Diplomarbeit am Lehrstuhl für Künstliche Intelligenz und Angewandte Informatik, Universität Würzburg, August 1998

[Ehren88]
A. S. C. Ehrenberg, Repeat-Buying: Facts, Theory and Application, Charles Griffin & Company Ltd., London 1988.

[FPSSU96]
U M. Fayyad, G Piatetsky-Shapiro, P Smyth, and Ramasamy Uthurusamy, Advances in Knowledge Discovery and Data Mining. AAAI Press / The MIT Press, Menlo Park, California, 1996.

[GeHa02]
Andreas Geyer-Schulz, Michael Hahsler, Evaluation of Recommender Algorithms for an Internet Information Broker based on Simple Association Rules and on the Repear-Buying Theory, in Proceedings WEBKDD 2002, Edmonton, Canada, July 2002

[Hahs02]
Michael Hahsler, A Simplified Model of Stationary Purchasing Behavior Applied to Information Goods on the Web, …. 2002

[HS93]
Maurice Houtsma and Arun Swami, Set-oriented mining for association rules. Technical report, IBM Research Division, Almaden Research Center, San Jose, CA, USA, 1993.

[Knuth81]
Donald Ervin Knuth: The Art of Computer Programming 2nd Edition, Addison-Wesley

[Marsa02]
Marsaglia George: Diehard, http://stat.fsu.edu/~geo/diehard.html Abruf am 1.10.2002

[MatNish98]
Makoto Matsumoto, Takuji Nischimura: Mersenne Twister: A 623-dimensionally equiditributed uniform pseudorandom number generator, Keio University, Yokohama 1998

[MoBy84]
Morgan, Byon: Elements of Simulation, Chatman & Hall, London - New York 1984

[NRINC92]
Numerical Recipes in C: the art of scientific Computing, Cambridge Press 1992.

[Quest02]
Quest, The Data Mining Group, http://www.almaden.ibm.com/cs/quest/HOME.html, Abruf vom 3.8.02

[RECHE00]
Peter Rechenberg, Was ist Informatik? Eine allgemeinverständliche Einführung, 2000

[RoCle96]
Robert T. Clemen: Making Hard Decisions, An Introduction to Decision Analysis 2nd Edition, Duxbury Press, Belmont 1996

[SRIAGR95]
Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. Technical report, IBM Research Division, Almaden Research Center, San Jose, CA, USA, 1995. Also in Proceedings of the 21st VLDB Conference 1995.

[SuhlL00]
Leena Suhl, Vorlesung Simulation webbasiertes Lernsystem, 2000

[Trenkl98]
Dietrich Trenkler, Die Erzeugung von Zufallszahlen mit MSExcel, Universität Osnabrück, Proseminar Statistik 1998

[ZhKoMa01]
Zijian Zheng, Ron Kohavi, Llew Mason, Real World Performance of Association Rule Algorithms, A short poster version to appear in KDD-2001, New York, 2001