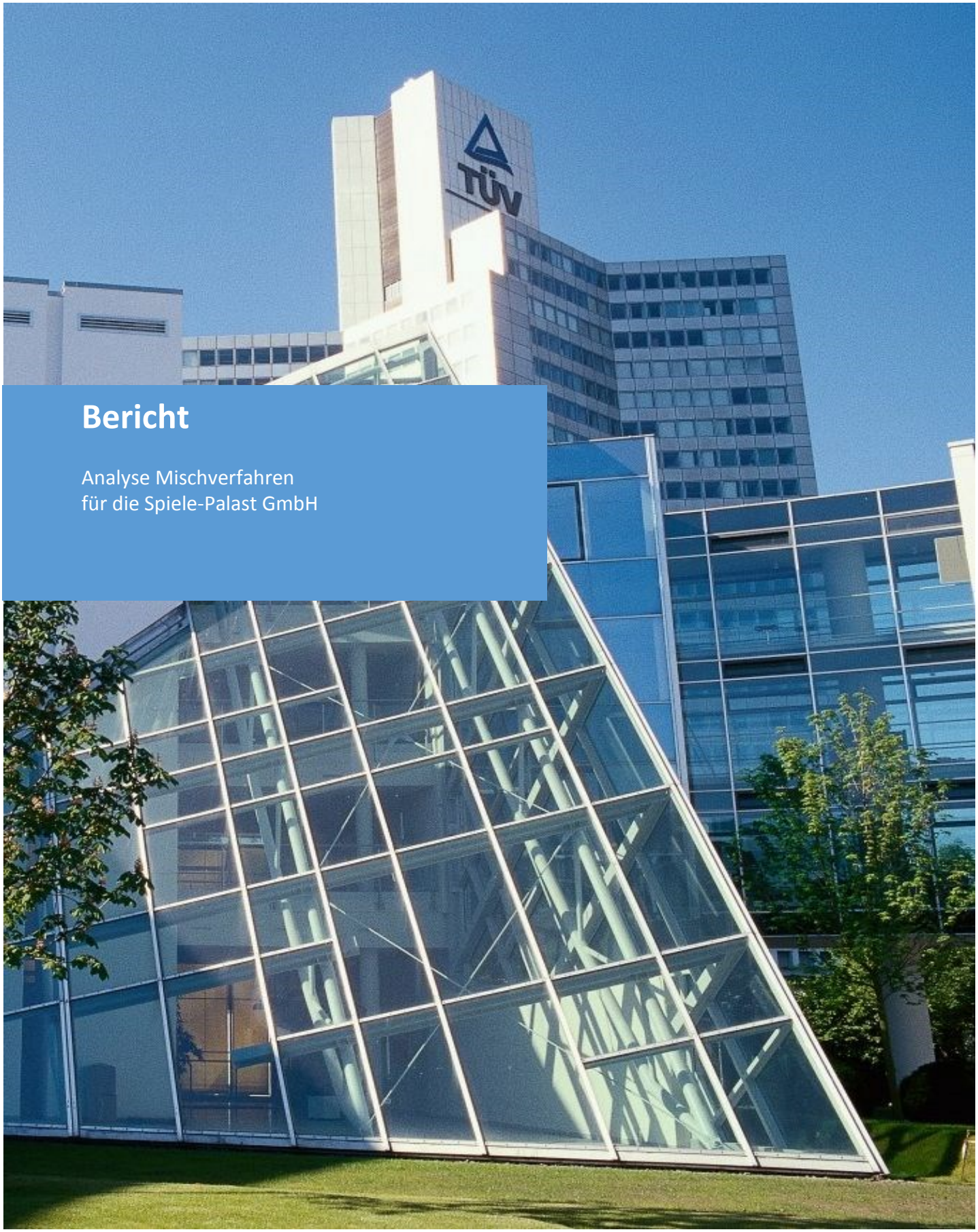
TÜVRheinland®
Genau. Richtig.



Bericht

Analyse Mischverfahren
für die Spiele-Palast GmbH

Allgemeine Information

Kundendaten

Spiele-Palast GmbH

Boxhagener Str. 106

10245 Berlin

Dienstleisterinformationen

TÜV Rheinland i-sec GmbH

TÜV Rheinland i-sec GmbH

Am Grauen Stein | 51105 Köln

Klassifikation

Dieses Dokument ist als „ÖFFENTLICH“ eingestuft.

Historie

Version	Modifikation	Datum	Typ
0.1	Erster Entwurf	27.05.2016	Entwurf
0.3	Analyse Mischverfahren	24.06.2016	Entwurf
0.5	Analyse Zufallszahlengenerator	25.08.2016	Entwurf
0.8	Analyse Geben/Platzierung	29.08.2016	Entwurf
0.9	Finaler Entwurf	31.08.2016	Entwurf
1.4	QA	20.09.2016	Entwurf
1.8	Finalisierung	21.09.2016	Final
1.9	Fehlerberichtigung	29.09.2016	Final
2.1	Überarbeitung zur Initialisierung	22.11.2016	Final
2.2	kleine Fehlerkorrekturen	27.01.2017	Final

1 Management-Zusammenfassung

Auf den zur Spiele-Palast GmbH (Spiele-Palast) gehörenden Webseiten werden verschiedene Online-Kartenspiele angeboten. Die diesem Angebot zugrunde liegende Software wird dabei von Spiele-Palast entwickelt und betrieben. Eine der Funktionen der Software simuliert das Mischen der Karten.

In der vorliegenden Untersuchung wird für sechs verschiedene Kartenspiele in insgesamt 13 Varianten die Qualität und Angemessenheit der Mischfunktion aus statistischer Sicht ermittelt und mit anderen Verfahren verglichen. Darüber hinaus wird die Frage geklärt, welchen Einfluss die Sitzposition eines Spielers in Bezug auf die Verteilung der Karten zu Beginn eines Spieles hat.

Die Analyse hat die folgenden Resultate ergeben.

(1) Das verwendete Verfahren mischt die Kartensätze schnell und gut. Der Mischalgorithmus ist in Verbindung mit dem verwendeten Pseudo-Zufallszahlengenerator geeignet, alle Permutationen eines Kartensatzes mit der gleichen Wahrscheinlichkeit zu erzeugen.
(2) Im Vergleich zu den üblichen Mischverfahren von Hand werden die Karten in der vorliegenden Implementierung deutlich besser durchmischt.
(3) Die Sitzposition beim Geben hat keinen Einfluss auf die Verteilung der Karten zu Beginn eines Spieles.

Alle Resultate gelten dabei aufgrund der analogen Implementierungen gleichwertig für alle untersuchten Varianten der Kartenspiele.

Im Vergleich zum Mischen von Hand wurden in der Implementierung einige Schritte ausgelassen. Dies beeinflusst an keiner Stelle das Mischergebnis oder die Fairness der Spielbeginne. Insbesondere haben die ausgelassenen Schritte aus statistischer Sicht keinen Nutzen.

2 Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

3 Hintergrund und Aufgabenstellung

Die Spiele-Palast GmbH (Spiele-Palast) bietet auf ihren Web-Seiten verschiedene Online-Kartenspiele an, die von ihr selbst entwickelt und betrieben werden. Die Spiele sind dabei jeweils nach den zugehörigen Regeln modelliert und simulieren deren Ablauf in Software. Spieler können so über das Internet gegeneinander antreten.

Eine der Funktionen der Software simuliert das Mischen der Karten. Dieses hat einen wesentlichen Einfluss auf einen fairen Beginn der jeweiligen Spiele. Nach den jeweiligen Spielregeln wird ein Durchmischen der Karten für den Beginn der einzelnen Spiele in unterschiedlichen Formen auch explizit gefordert (siehe z.B. [1]). Es hilft sicherzustellen, dass alle Spieler in Bezug auf die Verteilung der Karten im Spiel vergleichbar wenig Anfangswissen haben, da sie jeweils nur die Ihnen bekannt gemachten Blätter kennen können.

In der vorliegenden Untersuchung wird das Mischverfahren für sechs verschiedene Kartenspiele untersucht. Dazu gehören die vier auf den Webseiten von Spiele-Palast veröffentlichten Spiele:

- Skat,
- Mau-Mau,
- Doppelkopf,
- Schafkopf,

sowie zwei zum Zeitpunkt dieser Untersuchung noch nicht veröffentlichte Spiele

- Rommé und
- Solitaire.

Bei einigen der Spiele gibt es verschiedene Varianten mit einem oder zwei Blättern bzw. mit oder ohne zusätzliche Karten, sodass insgesamt 13 verschiedene Spielvarianten zu betrachten sind. Das Mischen der Spiele erfolgt dabei für alle Varianten immer auf Basis desselben Algorithmus.

In der vorliegenden Untersuchung wird für die Kartenspiele ermittelt,

(1) ob das der Mischfunktion zugrunde liegende Mischverfahren und der zugehörige Pseudo-Zufallszahlengenerator für den Zweck angemessen sind, das heißt, ob in einem statistischen Sinne „gut“ gemischt wird,
(2) wie das Mischverfahren in Bezug auf seine Güte im Vergleich zu anderen Mischverfahren einzuordnen ist, und
(3) welchen Einfluss die Sitzposition eines Spielers in Bezug auf das Verteilen der Karten und damit auf den Start eines Spieles hat.

3.1 Annahmen

Der Untersuchung werden die folgenden Annahmen zugrunde gelegt:

a) Die Mischfunktion ist in der Programmiersprache Java beschriebenen Algorithmus realisiert. Hierbei wird davon ausgegangen, dass auf technischer Ebene die Ausführung für ein beliebiges Online-Spiel entsprechend dem Quellcode fehlerfrei erfolgt. Dies umfasst
   - die Abarbeitung des Algorithmus,
   - die Verwaltung von zugehörigen Daten,
   - den Zugriff auf Datenbanken,
   - die Verwendung von fertigen Funktionsbibliotheken,
   - die Ein- und Ausgabe und
   - das Auftreten möglicher Seiteneffekte durch die Software oder andere Software auf den Systemen zur Webseite.

b) Die Initialisierung des Pseudo-Zufallszahlengenerators erfolgt unter Zurückgriff auf eine Java-Methode, die auf die Server-Hardware zurückgreift. Hier wird davon ausgegangen, dass diese fehlerfrei entsprechend den Spezifikationen funktioniert.

c) Die Verteilung der Karten in den verschiedenen Spielarten erfolgt analog zu der für das Skat-Spiel aufgezeigten Methode:

   Vom gemischten Stapel wird der Reihe nach die für das jeweilige Spiel notwendige Zahl von Karten für jeden Spieler bzw. für im Spiel genutzte weitere Kartenstapel genommen und an diese verteilt. Dies erfolgt dabei immer in der jeweils gleichen Weise und rein aufgrund der Position der Karte im gemischten Kartenstapel.

d) Es ergeben sich keine neuen Erkenntnisse zusätzlich zu den bekannten Stärken und Schwächen der den untersuchten Verfahren zugrunde liegenden Algorithmen.

4 Analyse

4.1 Vorgehensweise

Die Untersuchung ist in mehrere Schritte untergliedert, denen die Struktur dieses Dokuments folgt. Im folgenden Abschnitt 4.2 wird zunächst der gemeinsam genutzte Pseudo-Zufallszahlengenerator untersucht. Dieser bildet eine Grundlage für die in Abschnitt 4.3 betrachtete Mischfunktion. Abschnitt 4.4 widmet sich dem Einfluss der Sitzposition auf den Spielstart. In Abschnitt 4.5 werden schließlich die Ergebnisse zusammengefasst.

Für alle Abschnitte wird zunächst das untersuchte Thema vorgestellt und dann mit kurzer Begründung bewertet. Basis für alle Untersuchungen sind

- die Kurzbeschreibungen der Algorithmen [2],
- die zugehörigen Quellcode-Auszüge [3], [4] und [5], sowie
- die weiteren im Literaturverzeichnis in Abschnitt 5.2 angegebenen Dokumente. Diese werden an den jeweils genutzten Stellen aufgeführt.

Für die gesamte Vorgehensweise gelten die in Abschnitt 3.1 gelisteten Annahmen.

4.2 Generierung der Zufallszahlen

Grundlage der Mischfunktion für alle zu untersuchenden Spiele ist die Bereitstellung von Zufallszahlen, die den Verlauf der jeweiligen Mischvorgänge beeinflussen.

Hierzu wird von Spiele-Palast der mit einem Wert aus der der Java-Methode `SecureRandom()` (siehe [6]) initialisierte so genannte „Mersenne-Twister“ (siehe [7] bzw. Abschnitt 4.2.1 unten) genutzt.

In der Software wird dafür die Implementierung einer Open Source-Bibliothek der Apache Software Foundation verwendet (Siehe [8] bzw. Abschnitt 5.1.1), die eine von den beiden Autoren des „Mersenne-Twister“-Algorithmus im Jahre 2002 verbesserte vorgestellte Version in der Programmiersprache C in der Programmiersprache Java umsetzt.

Die folgende Abbildung zeigt dazu den Code-Teil aus der Server-Software, der auf die Apache-Implementierung zugreift (siehe Abschnitt 5.1.3), um den Generator zur Verwendung vorzubereiten.

```
038 //Due to the extremely long period of Mersenne Twister, the reseed method
is right now only called once: When the server is launching
039 public static void reseed() throws Exception
040 {
041   SecureRandom srnd = new SecureRandom();
042   long seed = 0;
043
044   do
045   {
046         seed = srnd.nextLong();
047   } while (seed == 0);
048
049   rand = new MersenneTwister(seed);
050
051   //Mersenne Twister Warmup
052   final byte[] dummy = new byte[1];
053   for (int i = 0; i < 800000; ++i)
054   {
055         rand.nextBytes(dummy);
056   }
057 }
```

Abbildung 1: Initialisierung des Pseudo-Zufallszahlengenerators

Die Initialisierung des „Mersenne-Twister“ erfolgt mit dem Ergebnis der Java-Methode `SecureRandom.nextLong()` als Initialisierungswert (Random Seed), siehe Zeile 46 bzw. 49. Danach werden zum „Aufwärmen“ des Generators $800\,000$ Pseudo-Zufallszahlen erzeugt (Zeilen 52-56).

4.2.1 Bewertung

Algorithmus

Der „Mersenne-Twister“ ist sehr gut untersucht und stellt einen mit wenig Aufwand in Software implementierbaren, schnellen und skalierbaren und allgemein anerkannten Algorithmus zur Erzeugung von Pseudo-Zufallszahlen dar (siehe [9]).

Mit Ausnahme des Tests auf lineare Unabhängigkeit – der Algorithmus basiert auf linearer Rekursion – bestehen die mit ihm erzeugten Zahlen alle Standard-Tests auf Zufälligkeit der einschlägigen Testsuiten, sofern der Generator hinreichend lang „eingeschwungen“ wurde (siehe [10]).

Die Periode von $2^{19937} - 1 \approx 4.315 \times 10^{6001}$ für die Wiederholung der Zahlen reicht für viele Anwendungen aus (siehe [7]). Insbesondere liegt sie deutlich über der Zahl der möglichen Spielbeginne für alle untersuchten Kartenspiele (siehe Tabelle 1, Spalte 3 auf Seite 9).

Für kryptographische Anwendungen ist der Algorithmus ungeeignet, denn eine Beobachtung von 624 erzeugten 32-Bit-Ergebnissen bzw. 19937 erzeugten 1-Bit-Ergebnissen genügt, um alle folgenden Pseudo-Zufallszahlen zu bestimmen [11]. Dieselbe Argumentation macht es damit im Prinzip möglich, mit dem Wissen über die genauen Startwerte hinreichend vieler Spiele und der jeweiligen Spieltypen alle weiteren folgenden Mischergebnisse vorherzusagen. Sie wird hier aber als nicht praktikabel angesehen:

> Ein solcher Angriff wäre, sofern sich keine Vereinfachungen finden lassen, mit einigem Aufwand und Insider-Wissen verbunden. Ein Angreifen müsste für mindestens 624 aufeinander folgende Kartenspiele
>
> - jeden Startwert für das Mischen bzw. die anfängliche Kartenverteilung, aus der dieser zurückgerechnet werden müsste, und
> - den Typ und die Variante des jeweiligen Kartenspieles
>
> kennen. Dies würde damit vermutlich einen Zugang zum Server-System erfordern, bei dem es deutlich praktikabler wäre, die Spiele auf andere Weise zu manipulieren.

Initialisierung

Für die Initialisierung des Generators wird wie oben beschrieben die Java-Methode `SecureRandom.nextLong()` genutzt. Diese erfüllt die Anforderungen des US-NIST [12] und erzeugt kryptographisch starke Zufallszahlenfolgen nach RFC 1750 [13] mit einer Entropie von 48 Bit [14].

Hierbei wird darauf geachtet (Zeilen 42-47), dass keine 0 als Initialisierungswert für den „Mersenne-Twister“ ausgewählt wird, denn diese würde aufgrund der Eigenschaften dieses Algorithmus (siehe [7]) dann nur die Nullfolge ergeben.

Hierzu lässt sich zunächst anmerken, dass

- die hierbei verwendete Implementierung – Durchlaufen einer Schleife so oft, bis ein von 0 verschiedener Zufallswert erzeugt worden ist – nicht sehr effizient ist und, dass
- ein vollständiger Durchlauf der Schleifen bis ein von 0 verschiedener Wert erreicht ist, theoretisch beliebig lange dauern kann – es könnten beliebig viele Nullen hintereinander erzeugt werden und es ist nicht ausgeschlossen, dass `SecureRandom.nextLong()` selbst eine unbestimmbare Zeit auf genügend Entropie „warten“ muss, um die nächste Zufallszahl auszugeben.

Beide Punkte hier werden allerdings nur als theoretische Ineffizienzen angesehen und beeinträchtigen nicht die „Güte“ des Mischverfahrens sondern die Startzeit des Servers.

Die oben beschriebene Periode des „Mersenne-Twisters“ ist mit den Angaben oben deutlich größer als die Zahl der möglichen Initialisierungswerte des „Mersenne-Twisters“, die bei $2^{48} \approx 2.81 \times 10^{14}$ liegt. Entsprechend der Dokumentation wir die Java-Methode einmal beim Start des Servers genutzt. Damit ergibt sich eine erreichbare maximale Entropie von H_S=48 Bit für den Start.

Tabelle 1 auf Seite 9 unten listet hierzu im Vergleich die erforderlichen Entropiewerte H_n, die jeweils sicherstellen, dass alle möglichen Blätter erzeugt werden können:

> Die Angaben in Spalte 2 stammen dabei aus den Berechnungen in den Zeilen 108, 139, 95, 82, 188, 176, 121, 162 bzw. 147 aus dem Quellcode in Abschnitt 5.1.3.
>
> Spalte 3 der Tabelle gibt einen gerundeten Wert für die jeweilige Zahl $|\mathcal{B}_n| = n!$ der Permutationen eines Blattes wieder. Die letzte Spalte der Tabelle enthält den gerundeten Logarithmus zur Bestimmung der erforderlichen Entropie $\mathrm{H}_n = \log_2|\mathcal{B}_n| = \log_2|n!|$ der Zahl n der möglichen Blätter, um sicherzustellen, dass durch das Mischen alle möglichen Blätter erreicht werden können.
>
> Der Zusatz „kurz“ bzw. „lang“ bei einigen Spielenamen in Spalte 1 steht in etwa dafür, ob ein bzw. zwei vollständige Kartensätze genutzt werden.

Spiel	Anzahl Karten n	Anzahl mögliche Blätter $\lvert\mathcal{B}_n\rvert$	erforderliche Entropie H_n
Schafkopf lang Skat Mau-Mau kurz	32	2.6×10^{35}	118 Bit
Schafkopf kurz	24	6.2×10^{23}	80 Bit
Doppelkopf ohne Neun	40	8.2×10^{47}	160 Bit
Doppelkopf	48	1.24×10^{61}	203 Bit
Rommé kurz ohne Joker Solitaire kurz	52	8.1×10^{67}	226 Bit
Rommé kurz	55	1.3×10^{76}	243 Bit
Mau-Mau lang	64	1.3×10^{89}	296 Bit
Rommé lang ohne Joker Solitaire lang	104	1.0×10^{166}	552 Bit
Rommé	110	1.6×10^{178}	592 Bit

Tabelle 1: Mögliche Blätter und erforderliche Entropie für die verschiedenen Spielvarianten

Ein Vergleich der erforderlichen mit der verfügbaren Entropie zeigt nun, dass die genutzte Implementierung jeweils nur einen sehr kleinen Teil der tatsächlichen Möglichkeiten für die jeweiligen Blätter der ein-

zelnen Spieltypen erzeugen kann. Damit ist es auch beim dem kleinsten Blatt und angenommener Entropie von 64 Bit für `SecureRandom.nextLong()` nicht möglich, alle denkbaren Initialisierungswerte und damit Spielbeginne zu erreichen; für alle n gilt $|\mathcal{B}_n| \gg 2^{64}$.

Dies ist eine technische Einschränkung der verwendeten Plattform. Ähnlich zu den Ausführungen im oberen Abschnitt ließe sich hieraus auch ein Angriff konstruieren. Aufgrund der jeweils großen Diskrepanz zwischen der Zahl der insgesamt möglichen und der Zahl der erreichbaren Blätter ließe sich mit statistischen Methoden ermitteln, welche Blätter durch ein Mischen *nicht* erreicht werden können – in allen Fällen ist dies aktuell die Mehrheit –, so dass sich ein Spieler hier einen Vorteil verschaffen kann.

Auch solch ein Angriff setzt allerdings ein großes Hintergrundwissen voraus und wird ähnlich wie der oben beschriebene Angriff gesehen.

Warmlaufen

Nach der Initialisierung wird der „Mersenne-Twister“, wie oben gezeigt, mit 800 000 Durchläufen „warm gelaufen“. Dies entspricht den empirischen Empfehlungen für diesen Fall, um eine gute Gleichverteilung der Pseudo-Zufallszahlen zu erhalten und sollte entsprechend so beibehalten werden (siehe [10]).

In Ergänzung hierzu könnte ein anderer Pseudo-Zufallszahlengenerator, wie z.B. Well19937a oder Well44497b verwendet werden, der keine entsprechen lange „Einschwingphasen“ benötigt (siehe ebenfalls [10]), sodass der Generator auch häufiger mit echten Zufallszahlen initialisiert werden kann.

Für die weitere Betrachtung kann damit nun vorausgesetzt werden:

Die verwendeten Zahlen sind gleichverteilt zufällig. **(A)**

4.3 Mischfunktion

Die Mischfunktion verwendet den Fisher-Yates-Algorithmus (siehe [15], [16]):

> Ausgehend vom Ende des Kartenstapels wird für jede im Blatt vorhandene Stelle einer Karte der Reihe nach ein Karte zufällig gewählt, weggelegt und von der nächsten Runde ausgeschlossen, in dem die zufällig ausgewählte Karte mit der Karte an der aktuellen Stelle vertauscht wird und die nächste auszuwählende Karte nur noch vom Rest der verbleibenden Karten ausgewählt wird.

Abbildung 2 zeigt hierzu die Implementierung von Spiele-Palast, die genau diese Vorgehensweise verwendet. Mit der Schleife in Zeile 218 wird das gesamte Blatt vom Ende her durchlaufen. In Zeile 220 werden jeweils die aktuelle Karte mit Position in der Laufvariablen `i` mit einer zufällig aus dem Rest, des verbleibenden Blattes ausgewählten Karte vertauscht, das heißt, mit der Karte an der mit Hilfe des „Mersenne-Twisters“ bestimmten Stelle `rand.nextInt(i + 1)`.[1]

```
214    private void shuffleFisherYates()
215    {
216          //Fisher-Yates Algorithm
217          //Traverse the list backwards up to 2nd element, swapping randomly
selected element from 0 to i into position of i
218          for (short i = (short) (size() - 1); i > 0; --i)
219          {
220                Collections.swap(this, i, rand.nextInt(i + 1));
```

[1] In der Methode `nextInt(int n)` wird dabei darauf geachtet, einen Modulo-Bias zu verhindern, siehe [27].

```
221             }
222
223     }
```

Abbildung 2: Implementierung des Mischalgorithmus

4.3.1 Bewertung

Im Folgenden wird zunächst kurz intuitiv die Unverfälschtheit des Mischalgorithmus begründet, das heißt die gleiche Wahrscheinlichkeit für alle Mischergebnisse:

Angenommen $\mathrm{B}=(k_1, k_2, \dots, k_n) \in \mathcal{B}_n$ ist ein Blatt aus n Karten $k_i, 1 \leq i \leq n$. Dabei sei der Wert $n \in \{32, 40, 48, 52, 55, 64, 104, 110\}$ abhängig vom jeweiligen Typ des Kartenspieles gewählt, und es sei dabei $\mathcal{B}_n$ die Menge aller Blätter mit n Karten.[2]

Ferner sei daraus $\mathrm{B}'=M_n(\mathrm{B}) = M_n(k_1, k_2, \dots, k_n) = (k'_1, k'_2, \dots, k'_n)$ das Blatt nach dem Mischvorgang M_n. Das Mischen selbst entspricht einer Permutation $M_n \epsilon \mathfrak{S}_n$)[3] mit $M_n: \mathcal{B}_n \mapsto \mathcal{B}_n$.

Von allen $|\mathcal{B}_n| = n!$ möglichen Permutationen[4] oder Mischvorgängen $M_j, 1 \leq j \leq n!$ haben dann nach dem am Anfang des Abschnittes beschriebenen Algorithmus genau $(n-1)!$ davon die Karte k_1 an der ersten Stelle, das heißt, sie wurde im ersten Durchlauf nicht bewegt. Derselben Argumentation folgend haben $(n-1)!$ weitere für den um 1 verkleinerten Stapel die Karte k_2 an der ersten Stelle, etc.

Anders ausgedrückt, durch den Algorithmus mit Ergebnis $\mathrm{B}'=(k'_1, k'_2, \dots, k'_n)$ wurde k'_1 aus der Menge $\{k_1, k_2, \dots, k_n\}$ gewählt, k'_2 aus der Menge $\{k_1, k_2, \dots, k_n\} \backslash \{k'_1\}$, k'_3 aus der Menge $\{k_1, k_2, \dots, k_n\} \backslash \{k'_1, k'_2\}$, etc.

Die n Stellen $s_i \in \{0, \dots, n-1\}, 1 \leq i \leq n$ der jeweils nächsten zu vertauschenden Karten wurden hierzu jeweils gleichverteilt zufällig (siehe **(A)**) und voneinander unabhängig gewählt:

Stelle s_0 stammt gleichverteilt zufällig aus der Menge $\{0, \dots, n-1\}$, Stelle s_1 stammt gleichverteilt zufällig aus $\{0, \dots, n-2\}$, etc. bis schließlich für die letzte betrachtete Stelle s_{n-1} immer nur ein Platz bleibt: $s_{n-1} \in \{0\}$. In der Implementierung wird daher sinnvollerweise auch nur bis zur Stelle `i > 0` gegangen (Code-Zeile 218).
Die gemeinsame Wahrscheinlichkeit p für ein n-Tupel $\{s_0, s_2, \dots, s_{n-1}\}$ aller Stellen beträgt damit insgesamt $p(\{s_0, s_1, \dots, s_{n-1}\}) = p(s_0) \times p(s_1) \times \cdots \times p(s_{n-1}) = \frac{1}{n} \times \frac{1}{n-1} \times \cdots \times 1 = \frac{1}{n!}$. Das entspricht aber genau $\frac{1}{|\mathcal{B}_n|}$, d.h., dem Kehrwert der Zahl der möglichen Blätter $\mathcal{B}_n$ bei n Karten. Der Wert p ist also in der zu erwartenden Weise und ausschließlich abhängig von der Zahl der Karten n eines Blattes B. Anders gesagt:

Jedes Mischergebnis ist gleich wahrscheinlich. **(B)**

---

[2] siehe Spalte 2 in Tabelle 1 auf Seite 9

[3] $\mathfrak{S}_n$ meint die Symmetrische Gruppe über einer n-elementigen Menge.

[4] siehe Spalte 3 in Tabelle 1 auf Seite 9

Die Ergebnisse des Fisher-Yates-Algorithmus sind in diesem Sinne unverfälscht.[5] Nach einem Durchlauf erreicht er damit eine Durchmischung, die in ihrer Qualität der der Zufallszahlen entspricht, d.h., die Position aller Karten hängt nur noch von diesen ab.

Die Laufzeitkomplexität des Verfahrens beträgt $\mathcal{O}(n)$, denn im Rahmen des Verfahrens müssen n mal je zwei Elemente eines Feldes mit jeweils fester Größe vertauscht werden, was einen konstanten Aufwand erfordert.

In Bezug auf die Kartenspiele ist damit gewährleistet, dass immer schnell und „gut" gemischt wird.

Einordnung

Wie in der Einleitung zu 4.3 beschrieben bzw. im letzten Abschnitt noch einmal rigider gezeigt, entspricht damit das einmalige Mischen mit dem Fisher-Yates-Algorithmus dem zufälligen und für alle möglichen Ergebnisse gleichwahrscheinlichen Neuordnen eines Blattes, in dem aus einem bestehenden Blatt jeweils zufällig eine Karte ausgewählt und beiseite gelegt wird, bis alle Karten aufgebraucht sind.

In der folgenden Tabelle sind hierzu im Vergleich einige übliche Mischmethoden zusammengestellt, die beim Kartenspielen Anwendung finden (siehe dazu auch [17]).

Name	Methode
Durchwühlen	Ein Kartenstapel wird mit den Bildern nach unten auf einem Tisch ausgebreitet und mit den Händen in kreisförmigen Bewegungen durchmischt.
Überhand-Mischen	Mit einer Hand werden vom Kartenstapel in der zweiten Hand kleine Päckchen abgezogen und zu einem neuen kombiniert. Dies ist eine der gängigsten Mischmethoden in Deutschland.
Riffeln	Ein Kartenstapel wird in zwei in etwa gleich große Teilstapel geteilt, die dann mit Hilfe des Daumens nach oben gewölbt und durch gleichzeitiges Loslassen ineinander verzahnt werden.
Striping	Es werden kleine Teilstapel oben und unten vom Kartenstapel entfernt und so zu einem neuen Stapel zusammengesetzt.
Ineinanderdrücken oder Fächern	Ein Kartenstapel wird in etwa gleich große Teilstapel geteilt und gegeneinandergedrückt bzw. zuvor aufgefächert und dann ineinander geschoben.
Abheben	Von einem gemischten Kartenstapel werden ein oder mehrere kleinere Teilstapel von oben entfernt und in anderer als der ursprünglichen Reihenfolge zusammengesetzt.

Tabelle 2: Übliche Mischverfahren

Keine dabei entspricht dem Fisher-Yates-Algorithmus. Verglichen mit diesen Verfahren ließe sich die von Spiele-Palast verwendete Vorgehensweise mit einem modifizierten „Durchwühlen" vergleichen:

> Die Spielkarten werden mit dem Gesicht nach unten auf einem Tisch ausgebreitet und mit kreisenden Bewegungen untereinander vermischt und anschließend wieder aufgesammelt.
>
> Bei hinreichend gutem Durchmischen mit der Hand und dem sukzessiven Zusammenstellen des neu gemischten Blattes Karte für Karte, während die verbleibenden Karten weiter durchwühlt werden, ließe sich so intuitiv verglichen ein Ergebnis ähnlich dem Fisher-Yates-Mischen erzeugen.

[5] Die ausgelassen Punkte in der Argumentation hier lassen sich mittels vollständiger Induktion einfach ergänzen.

Tendenziell ist dies aber aus statistischer Sicht als schlechter durchmischt zu erwarten:

> Zum Einen bleiben beim Durchwühlen des Blattes Karten, die zu Beginn des Vorganges nah beieinander waren, aufgrund von physikalischen Effekten wie Haftreibung, elektrostatischer Anziehung oder Adhäsion während des Durchwühlens tendenziell ebenfalls nah beieinander. Zum Anderen hängt die Auswahl der jeweils nächsten zu entnehmenden Karte vom Mischenden und dessen Fähigkeit ab, eine Karte zufällig auszuwählen ab, sodass sich hier ein weiterer Bias zu erwarten ist.

Für die oben beschriebenen Verfahren wurden zahlreiche Untersuchungen zu deren „Güte“ durchgeführt, dabei auch zur Zahl der Durchläufe, bis ein hinreichend durchmischter Kartenstapel erreicht ist (siehe [18], [19], [20], [21], [22], [23], [24] oder [25]).

Allgemein werden in diesen Dokumenten, abhängig vom

- jeweiligen Verfahren,
- der Zahl der zu mischenden Karten, und
- der Frage, ob die Farben der Spielkarten auch „gut“ gemischt sein sollen,

immer mehr als einer und im Allgemeinen mindestens 5 bis 7 oder mehr Durchläufe empfohlen. Auch dies ist intuitiv klar, da bei jedem der Algorithmen oben Karten nicht vollständig unabhängig voneinander vermischt werden bzw. der Mischende eine Auswahl treffen muss.

Nach Feststellung **(B)** ist eine mehrfache Anwendung der Fisher-Yates-Algorithmus für die vorliegenden Implementierung nicht notwendig und würde keinen Vorteil in Bezug auf die Durchmischung erreichen.

In einigen der Regelwerke für die simulierten Kartenspiele kann es nun allerdings vorkommen, dass in Bezug auf den Umgang mit Karten bestimmte Verfahren verlangt werden [1]. So kann ein Abheben vor allem sicherstellen, dass bei einem realen Spiel niemand die letzte Karte kennen kann.

Aus statistischer Sicht und für am Computer simulierte Spiele ist eine solche Regel mit dem oben gesagten nicht notwendig und bringt hier weder eine bessere Durchmischung noch eine Veränderung in der Fairness der jeweiligen Spiele, da auch der Kartenstapel beim Mischen nicht eingesehen werden kann.

Muss eines der in Tabelle 2 aufgeführten Verfahren im Sinne von „guter“ Durchmischung häufiger angewandt werden, um eine bestimmte „Güte“ zu erreichen, so ist dies bei Verwendung des Fisher-Yates-Algorithmus, wie gerade dargestellt, nutzlos.

Zusammenfassend lässt sich feststellen:

> Die implementierte Mischroutine durchmischt die Karten besser als alle Mischverfahren von Hand. **(C)**

4.4 Einfluss der Sitzposition beim Geben

In Abschnitt 5.1.4 wird anhand eines Skat-Spieles das Geben der Karten gezeigt:

> Nacheinander bekommen der erste, der zweite und der dritte Spieler jeweils 10 Karten aus dem gemischten Kartenstapel; der Rest kommt in den Skat.

Für die vorliegende Betrachtung wird davon ausgegangen, dass dies auch für die anderen Kartenspiele der Fall ist (siehe Abschnitt 3.1c)).

4.4.1 Bewertung

Nach den Vorarbeiten in Abschnitt 4.2 und 4.3 lässt sich nun schnell beurteilen, welchen Einfluss die Sitzposition auf den Start eines Spieles hat:

Wie oben beschrieben – siehe Feststellung **(B)** – kann nun von einem zufälligen Mischergebnis für ein Blatt $B'=(k'_1, k'_2, \dots, k'_n)$ ausgegangen werden.

Durch das Geben der Karten wird im Falle von Skat ein gemischtes Blatt $B'=(k'_1,\dots, k'_{32})$ in die vier Stapel $B_{VH}=(k'_1, k'_2, \dots, k'_{10})$, $B_{MH}=(k'_{11}, k'_{12}, \dots, k'_{20})$, $B_{HH}=(k'_{21}, k'_{22}, \dots, k'_{30})$ und $B_S=(k'_{31}, k'_{32})$ für Vorhand, Mittelhand, Hinterhand und Skat aufgeteilt: $B' = B_{VH} \cup B_{MH} \cup B_{HH} \cup B_S$.

Das Geben erfolgt dabei immer nach demselben Muster und allein aufgrund der Position der Karten in B'. Seien nun die Karten nicht sofort direkt herausgegeben, sondern es werde zuvor den Stellen, an denen sie im Stapel liegen, eine allein von der ursprünglichen Bezeichnung der Stelle abhängige, dem jeweiligen Spiel entsprechende neue Bezeichnung gegeben, die das jeweilige Ziel der Ausgabe bezeichnen, im Beispiel Skat also entsprechend B_{VH}, B_{MH}, B_{HH} und B_s.

In Bezug auf die hier entwickelte Nomenklatur wird mithin der jeweilige Wert des Index i, $1 \leq i \leq n$ für die Karten k_i geändert. Die Karten selbst werden weder verändert, noch vertauscht.

Für den genutzten Algorithmus beim Skat-Spiel könnte die Umbenennung der Indizes also wie in der folgenden Tabelle 3 dargestellt geschehen.[6]

alt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
neu	$VH1$	$VH2$	$VH3$	$VH4$	$VH5$	$VH6$	$VH7$	$VH8$	$VH9$	$VH10$	$MH1$	$MH2$	$MH3$	$MH4$	$MH5$	$MH6$
alt	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
neu	$MH7$	$MH8$	$MH9$	$MH10$	$HH1$	$HH2$	$HH3$	$HH4$	$HH5$	$HH6$	$HH7$	$HH8$	$HH9$	$HH10$	$S1$	$S2$

Tabelle 3: Kartengeben - Beispiel

Die Wahrscheinlichkeiten für die einzelnen Karten k'_i an der entsprechenden Stelle zu sein, ändern sich damit durch das Geben, d.h., das Umbenennen von deren Positionen nicht. Dasselbe gilt für ein Geben der Karten nach den üblichen Skat-Regeln, das in Tabelle 4 dargestellt ist.

alt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
neu	$VH1$	$VH2$	$VH3$	$MH1$	$MH2$	$MH3$	$HH1$	$HH2$	$HH3$	$S1$	$S2$	$VH4$	$VH5$	$VH6$	$VH7$	$MH4$
alt	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
neu	$MH5$	$MH6$	$MH7$	$HH4$	$HH5$	$HH6$	$HH7$	$VH8$	$VH9$	$VH10$	$MH8$	$MH9$	$MH10$	$HH8$	$HH9$	$HH10$

Tabelle 4: Kartengeben - Beispiel Skat

Wird nun davon ausgegangen, dass die anderen Verteilalgorithmen nach demselben Muster verlaufen und damit dieselbe Argumentation angewandt werden kann, lässt sich zusammenfassen:

> Die Sitzposition beim Gehen hat keinen Einfluss auf die Verteilung der Karten zu Beginn eines Spieles. **(D)**

Hierzu sei noch einmal angemerkt, dass sich **(D)** auf den *Beginn* des Spieles bezieht. Welche Einflüsse die Sitzposition auf den *Verlauf* der verschiedenen Kartenspiele haben, lässt sich nicht so einfach sagen und ist nicht Gegenstand der Untersuchung:

> Hier ist nicht auszuschließen, dass die Sitzposition bei den verschiedenen Spielen generell Vor- oder Nachteile hat. So kann zum Beispiel im Spiel „Tic-Tac-Toe“ durch den ersten Spieler am Zug

[6] Hierbei stehen „alt“ und „neu“ für den ursprünglichen bzw. den neu zugewiesenen Index, VH, MH, HH und S jeweils wie oben wieder für Vorhand, Mittelhand, Hinterhand und Skat und die Zahlen dahinter für die laufende Nummer der verteilten Karte.

immer ein Unentschieden erreicht werden [26], bei „Vier Gewinnt“ kann der erste Spieler am Zug immer gewinnen [27] oder bei „Nim“, hängt der Gewinner von der Anfangskonstellation ab, sofern die jeweiligen Spieler einem festen Algorithmus folgen [28].

Ähnliche Aussagen könnten für einige der Kartenspiele zutreffen. Damit diese aber universell gelten können, müssen sie unabhängig von der Verteilung der Karten zu Beginn des Spieles und mit dem oben gesagten damit auch vom Geben der Karten sein.

Für die vorliegende Untersuchung lässt sich damit insgesamt feststellen, dass ein möglicher Einfluss auf den Spielverlauf unabhängig davon ist, ob das Spiel online oder mit echten Karten gespielt wird, sondern nur von den befolgten Regeln abhängt.

4.5 Ergebnisse

Im Rahmen der vorliegenden Analyse wurden die drei in Abschnitt 3 auf Seite 5 beschriebenen Fragestellungen untersucht. Dabei wurden die folgenden Resultate ergeben:

Zu (1): Das hier untersuchte Verfahren mischt die Kartensätze aus den Online-Kartenspielen effizient und erzeugt dabei gleich wahrscheinliche Ergebnisse. Der hierfür verwendete „Mersenne-Twister“ generiert für das Verfahren sehr gut geeignete gleichverteilte Zufallszahlen als Basis. In diesem Sinne ist die Vorgehensweise zum Mischen der Karten in der Kombination der beiden Verfahren als angemessen anzusehen; die simulierten Karten werden für alle Online-Kartenspiele „gut“ gemischt.

Ein betrachteter möglicher Angriff auf das Verfahren stellt sich als nicht praktikabel heraus.

Zu (2): Der benutzte Algorithmus entspricht keinem der üblichen Verfahren für das Mischen von Karten mit der Hand genau, lässt sich aber von Hand simulieren. Die in einigen Regelwerken herangezogenen üblichen Verfahren für das Mischen von Hand erreichen wesentlich langsamer eine Durchmischung der Karten.

Damit ist der verwendete Mischalgorithmus aus statistischer Sicht als deutlich besser und sinnvoller einzustufen als die Verfahren von Hand.

Die in der vorliegenden Implementierung ausgelassenen Schritte im Vergleich zum Mischen von Hand sind aus Sicht der Implementierung auf einem Computer wirkungslos und stellen somit für keinen der Spieler einen Nachteil dar.

Zu (3): Die Sitzposition eines Spielers wie auch der genaue Verteilungsalgorithmus beim Geben der Karten haben keinen Einfluss auf die Verteilung der Karten zu Beginn eines Spieles. Auch hier sind die in der vorliegenden Implementierung ausgelassenen Schritte im Vergleich zum Vorgehen von Hand aus statistischer Sicht wirkungslos. Ihr Weglassen stellt für keinen der Spieler einen Nachteil dar.

5 Anhang

5.1 A – Quellcode

5.1.1 Mersenne-Twister

Der folgende Quellcode zeigt den Kern des benutzten Pseudo-Zufallszahlengenerators (siehe [8]).

```
01 /** Generate next pseudorandom number.
02 * <p>This method is the core generation algorithm. It is used by all the
03 * public generation methods for the various primitive types {@link
04 * #nextBoolean()}, {@link #nextBytes(byte[])}, {@link #nextDouble()},
05 * {@link #nextFloat()}, {@link #nextGaussian()}, {@link #nextInt()},
06 * {@link #next(int)} and {@link #nextLong()}.</p>
07 * @param bits number of random bits to produce
08 * @return random bits generated
09 */
10 @Override
11 protected int next(int bits) {
12
13 int y;
14
15 if (mti >= N) { // generate N words at one time
16 int mtNext = mt[0];
17 for (int k = 0; k < N - M; ++k) {
18 int mtCurr = mtNext;
19 mtNext = mt[k + 1];
20 y = (mtCurr & 0x80000000) | (mtNext & 0x7fffffff);
21 mt[k] = mt[k + M] ^ (y >>> 1) ^ MAG01[y & 0x1];
22 }
23 for (int k = N - M; k < N - 1; ++k) {
24 int mtCurr = mtNext;
25 mtNext = mt[k + 1];
26 y = (mtCurr & 0x80000000) | (mtNext & 0x7fffffff);
27 mt[k] = mt[k + (M - N)] ^ (y >>> 1) ^ MAG01[y & 0x1];
28 }
29 y = (mtNext & 0x80000000) | (mt[0] & 0x7fffffff);
30 mt[N - 1] = mt[M - 1] ^ (y >>> 1) ^ MAG01[y & 0x1];
31
32 mti = 0;
33 }
34
35 y = mt[mti++];
36
37 // tempering
38 y ^=  y >>> 11;
39 y ^= (y <<   7) & 0x9d2c5680;
40 y ^= (y <<  15) & 0xefc60000;
41 y ^=  y >>> 18;
42
43 return y >>> (32 - bits);
44
45 }
```

5.1.2 Kartenarten

Der folgende Quellcode zeigt die Klassendefinition für die Spielkarten (aus [3]).

```
01 package de.rgerlach.game;
02
03 public class Card {
04    public enum Color
05    {
06          DIAMONDS,
07          HEARTS,
08          SPADES,
09          CLUBS
10    }
11
12    public enum Value
13    {
14          ACE,
15          TWO,
16          THREE,
17          FOUR,
18          FIVE,
19          SIX,
20          SEVEN,
21          EIGHT,
22          NINE,
23          TEN,
24          JACK,
25          QUEEN,
26          KING,
27          JOKER
28    }
29
30    private Color color;
31    private Value value;
32
33    public Card(Color _color, Value _value)
34    {
35          color = _color;
36          value = _value;
37    }
38
39    public Color color()
40    {
41          return color;
42    }
43
44    public Value value()
45    {
46          return value;
47    }
48
49    public String toString() {
50          return color + " : " + value;
51    }
52
53    public String encode() throws Exception
54    {
```

```
55          return String.valueOf(color().ordinal()) + (value().ordinal() < 10
? "0" : "") + String.valueOf(value().ordinal());
56      }
57  }
```

5.1.3 Quellcode – Blatt erzeugen

Der folgende Quellcode zeigt das Erzeugen der Blätter für die jeweiligen Spielarten mit dem Mischalgorithmus als Kern aus [3] bzw. [4].

```
001 package de.rgerlach.game;
002
003 import java.util.Collections;
004 import java.util.LinkedList;
005
006 import org.apache.commons.math3.random.MersenneTwister;
007
008 import de.rgerlach.game.Card.Color;
009 import de.rgerlach.game.Card.Value;
010
011 public class CardDeck extends LinkedList<Card>
012 {
013   /**
014    *
015    */
016   private static final long serialVersionUID = 157564870328490793L;
017
018   public enum DECK_TYPE
019   {
020         DOPPELKOPF,
021         DOPPELKOPF_NO_NINES,
022         SKAT,
023         SCHAFKOPF_LONG,
024         SCHAFKOPF_SHORT,
025         MAUMAU_LONG,
026         MAUMAU_SHORT,
027         ROMME_LONG,
028         ROMME_LONG_NO_JOKERS,
029         ROMME_SHORT,
030         ROMME_SHORT_NO_JOKERS,
031         SOLITAIRE_LONG,
032         SOLITAIRE_SHORT;
033   }
034
035   private static MersenneTwister rand = null;
036
037
038   //Due to the extremely long period of Mersenne Twister, the reseed
method is right now only called once: When the server is launching
039   public static void reseed() throws Exception
040   {
041         SecureRandom srnd = new SecureRandom();
042         long seed = 0;
043
044         do
045         {
046               seed = srnd.nextLong();
047         } while (seed == 0);
048
049         rand = new MersenneTwister(seed);
050
051         //Mersenne Twister Warmup
052         final byte[] dummy = new byte[1];
```

```
053         for (int i = 0; i < 800000; ++i)
054         {
055               rand.nextBytes(dummy);
056         }
057   }
058
059   private int nextCard;
060
061   /*
062   The constructor of the card deck:
063
064   The class CardDeck is instantiated once per table. A table consists of
a specific GameType, a number of players and a number of rounds.
065   For example:
066
067   GameType: SKAT
068   Number of players: 3
069   Number of rounds: 9
070
071   The CardDeck is then shuffled at the beginning of each round by calling
shuffle()
072   and players receive a number of cards by calling CardDeck.nextCard()
073
074   */
075   public CardDeck(DECK_TYPE type) throws Exception
076   {
077         nextCard = 0;
078
079
080         switch(type)
081         {
082         //cards from NINE to KING plus ACE, each card twice (48 total)
083         case DOPPELKOPF:
084               for (int h = 0; h < 2; ++h)
085                     for (int i = 0; i < 4; ++i)
086                     {
087                           for (int j = Value.NINE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
088                                 add(new Card(Color.values()[i],
Value.values()[j]));
089
090                           add(new Card(Color.values()[i], Value.ACE));
091                     }
092
093               break;
094
095         //cards from TEN to KING plus ACE, each card twice (40 total)
096         case DOPPELKOPF_NO_NINES:
097               for (int h = 0; h < 2; ++h)
098                     for (int i = 0; i < 4; ++i)
099                     {
100                           for (int j = Value.TEN.ordinal(); j <
Value.JOKER.ordinal(); ++j)
101                                 add(new Card(Color.values()[i],
Value.values()[j]));
102
103                           add(new Card(Color.values()[i], Value.ACE));
104                     }
```

```
105
106                 break;
107
108           //cards from SEVEN to KING plus ACE (32 total)
109           case SCHAFKOPF_LONG:
110           case SKAT:
111           case MAUMAU_SHORT:
112                 for (int i = 0; i < 4; ++i)
113                 {
114                       for (int j = Value.SEVEN.ordinal(); j <
Value.JOKER.ordinal(); ++j)
115                             add(new Card(Color.values()[i], Value.val-
ues()[j]));
116
117                       add(new Card(Color.values()[i], Value.ACE));
118                 }
119           break;
120
121           //cards from SEVEN to KING plus ACE, each card twice (64 total)
122           case MAUMAU_LONG:
123                 for(int k=0; k<2; k++)
124                 {
125                       for (int i = 0; i < 4; ++i)
126                       {
127                             for (int j = Value.SEVEN.ordinal(); j <
Value.JOKER.ordinal(); ++j)
128                                   add(new Card(Color.values()[i],
Value.values()[j]));
129
130                             add(new Card(Color.values()[i], Value.ACE));
131                       }
132                 }
133           break;
134
135           //cards from NINE to KING plus ACE (36 total)
136           case SCHAFKOPF_SHORT:
137                 for (int i = 0; i < 4; ++i)
138                 {
139                       for (int j = Value.NINE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
140                             add(new Card(Color.values()[i], Value.val-
ues()[j]));
141
142                       add(new Card(Color.values()[i], Value.ACE));
143                 }
144
145           break;
146
147           //cards from ACE to KING plus 3 JOKERS, each card twice (110 to-
tal)
148           case ROMME_LONG:
149                 for (int h = 0; h < 2; ++h)
150                 {
151                       for (int i = 0; i < 4; ++i)
152                       {
153                             for (int j = Value.ACE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
```

```
154                                     add(new Card(Color.values()[i],
Value.values()[j]));
155                         }
156                         for (int i = 1; i < 4; ++i)
157                               add(new Card(Color.values()[i], Value.JOKER));
                            // every color except diamonds
158                 }
159
160           break;
161
162           //cards from ACE to KING, each card twice (104 total)
163           case ROMME_LONG_NO_JOKERS:
164           case SOLITAIRE_LONG:
165                 for (int h = 0; h < 2; ++h)
166                 {
167                       for (int i = 0; i < 4; ++i)
168                       {
169                             for (int j = Value.ACE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
170                                     add(new Card(Color.values()[i],
Value.values()[j]));
171                       }
172                 }
173
174           break;
175
176           //cards from ACE to KING plus 3 JOKERS (55 total)
177           case ROMME_SHORT:
178                 for (int i = 0; i < 4; ++i)
179                 {
180                       for (int j = Value.ACE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
181                             add(new Card(Color.values()[i], Value.val-
ues()[j]));
182                 }
183                 for (int i = 1; i < 4; ++i)
184                       add(new Card(Color.values()[i], Value.JOKER));
                    // every color except diamonds
185
186           break;
187
188           //cards from ACE to KING (52 total)
189           case ROMME_SHORT_NO_JOKERS:
190           case SOLITAIRE_SHORT:
191                 for (int i = 0; i < 4; ++i)
192                 {
193                       for (int j = Value.ACE.ordinal(); j <
Value.JOKER.ordinal(); ++j)
194                             add(new Card(Color.values()[i], Value.val-
ues()[j]));
195                 }
196
197           break;
198           }
199     }
200
201     public void shuffle() throws Exception
202     {
```

```
203         nextCard = 0;
204         synchronized(rand)
205         {
206               shuffleFisherYates();
207         }
208   }
209
210   public Card nextCard() throws Exception
211   {
212         Card c = get(nextCard++);
213         return c;
214   }
215
216   public boolean hasNext()
217   {
218         return nextCard < this.size();
219   }
220
221
222   private void shuffleFisherYates()
223   {
224         //Fisher-Yates Algorithm
225         //Traverse the list backwards up to 2nd element, swapping ran-
domly selected element from 0 to i into position of i
226         for (short i = (short) (size() - 1); i > 0; --i)
227         {
228               Collections.swap(this, i, rand.nextInt(i + 1));
229         }
230
231   }
232 }
```

5.1.4 Pseudocode – Kartengeben Skat

Der folgende Pseudocode zeigt beispielhaft das Geben der Karten für das Skat-Spiel (siehe [5]).

```
01  protected void prepareNextRound() throws Exception
02  {
03    _deck.shuffle();
04
05    for (int i = _vorhand; i < _vorhand + 3; ++i)
06    {
07          SkatPlayer player = (SkatPlayer) _players.get(i % 3);
08
09          // clear stack and hand
10          player.stack().clear();
11          player.hand().clear();
12
13          // add cards
14          for (int j = 0; j < _deck.size() / 3; ++j)
15          {
16                Card card = _deck.nextCard();
17                player.addCard(card);
18          }
19
20          //send cards to player, etc
```

```
21          //...
22      }
23
24      //add remaining 2 cards into the Skat
25      _skat = new SkatStapel(_deck.nextCard(), _deck.nextCard());
26  }
```

5.2 B – Verweise

[1] Deutscher Online Skatverband, „Internationale Online Skatordnung," 31 März 2008. [Online]. Available: http://www.doskv.de/iosko.pdf. [Zugriff am 1 Juli 2016].

[2] R. Gerlach, *Spiele-Palast Mischalgorithmus,* 2016.

[3] R. Gerlach, *Card.java,* 2016.

[4] R. Gerlach, *CardDeck.java,* 2016.

[5] R. Gerlach, *SkatGameExample.pseudo.java,* 2016.

[6] Oracle Corporation, „JavaTM Platform Standard Ed. 8," 1993, 2016. [Online]. Available: http://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html. [Zugriff am 1 Juli 2016].

[7] M. Matsumoto und T. Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator,* Bd. 8, 1998, pp. 3-30.

[8] The Apache Software Foundation, „Mersenne Twister (Apache Commons Math 3.5 API)," 2016. [Online]. Available: http://commons.apache.org/proper/commons-math/javadocs/api-3.5/src-html/org/apache/commons/math3/random/MersenneTwister.html. [Zugriff am 1 Juli 2016].

[9] P. L'Ecuyer und R. Simard, „TestU01: A C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS),* Bd. 33, Nr. 4, August 2007.

[10] F. Panneton, P. L'Ecuyer und M. Matsumoto, „Improved Long-Period Generators Based on Linear Recurrences Modulo 2," *ACM Transactions on Mathematical Software,* Bd. 32, Nr. 1, pp. 1-16, 2006.

[11] „Frequently asked quesitions," 2016. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/efaq.html. [Zugriff am Juli 2016].

[12] National Institute of Standards and Technology, „Computer Security Division / Computer Security Resource Center," 25 Mai 2001. [Online]. Available: http://csrc.nist.gov/groups/STM/cmvp/standards.html#02. [Zugriff am 1 Juli 2016].

[13] Network Working Group, Internet Engineering Task Force, Dezember 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1750.txt. [Zugriff am 1 Juli 2016].

[14] Oracle Corporation, „Java Platform Standard Ed. 8," 1993, 2016. [Online]. Available: http://docs.oracle.com/javase/8/docs/api/java/util/Random.html#nextLong--. [Zugriff am 1 Juli 2016].

[15] P. E. Black, „Fisher-Yates shuffle," National Institute of Standards and Technology, 9 March 2015. [Online]. Available: http://www.nist.gov/dads/HTML/fisherYatesShuffle.html. [Zugriff am 1 Juli 2016].

[16] D. E. Knuth, „Seminumerical algorithms," in *The Art of Computer Programming*, 2nd Hrsg., Bd. 2, Reading, Massachusetts, Addison-Wesley Publishing Company, 1964, pp. 139-140.

[17] Wikipedia, „Mischen (Spielkarten)," 2016. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Mischen_(Spielkarten)&oldid=144784288. [Zugriff am 1 Juli 2016].

[18] D. Aldous und P. Diaconis, „Shuffling Cards and Stopping Times," *Amer. Math. Monthly,* Bd. 93, pp. 333-348, 1986.

[19] L. N. Trefethen, „How Many Shuffles to Randomize a Deck of Cards?," *Proceedings of the Royal Society,* Bd. 456, Nr. 2002, pp. 2561-2568, 8 October 2000.

[20] A. van Zuylen und F. Schalekamp, „The Achilles' Heel of the GSR shufle: A Note on New Age Solitaire," *Probability in the Engineering and Informational Sciences,* Bd. 18 (3), pp. 315-328, 2008.

[21] B. Mann, „How many times should you shuffle a deck of cards?," *Topics in contemporary probability and its applications,* pp. 261-289, 1995.

[22] G. Kolata, „In Shuffling Cards, 7 Is Winning Number," 9 Januar 1990. [Online]. Available: http://www.nytimes.com/1990/01/09/science/in-shuffling-cards-7-is-winning-number.html. [Zugriff am 1 Juli 2016].

[23] D. Goldstein und D. Moews, „The Identity Is The Most Likely Exchange Shuffle For Large n," 06 10 2000. [Online]. Available: http://arxiv.org/abs/math.CO/0010066. [Zugriff am 08 2011].

[24] J. Jonasson, „The Overhand Shuffle Mixes In $\Theta(N^2 \log N)$ Steps," *The Annals of Applied Probability,* Bd. 16, Nr. 1, pp. 231-243, 2006.

[25] R. Permantle, „Randomization Time for the Overhand Shuffle," *Journal of Theoretical Probahility,* Bd. 3, Nr. 1, 1989.

[26] E. W. Weisstein, „Wolfram Mathworld--A Wolfram Web Resource," 2016. [Online]. Available: http://mathworld.wolfram.com/Tic-Tac-Toe.html. [Zugriff am 1 Juli 2016].

[27] V. Allis, *A Knowledge-based Approach of Connect-Four, The Game is Solved: White Wins,* V. U. Department of Mathematics and Computer Science, Hrsg., Amsterdam, The Netherlands, 1988.

[28] C. L. Bouton, *Nim, a game with a complete mathematical theory,* Bd. 3 No. 1/4, 1901-1902, pp. 35-39.

[29] P. Hellekalek, „Good random number generators are (not so) easy to find," *Mathematics and Computers in Simulation,* Bd. 46, pp. 485-505, 1998.

[30] Oracle Corporation, „Java™ Platform Standard Ed. 8," 1993, 2016. [Online]. Available: http://docs.oracle.com/javase/8/docs/api/index.html. [Zugriff am 1 Juli 2016].

[31] The Apache Software Foundation, „BitStreamGenerator (Apache Commons Math 3.5 API)," 2016. [Online]. Available: http://commons.apache.org/proper/commons-math/javadocs/api-3.5/org/apache/commons/math3/random/BitsStreamGenerator.html#nextInt(int). [Zugriff am 1 Juli 2016].

5.3 C – Allgemeine Hinweise

Im Hinblick auf den Charakter der Untersuchung als Betrachtung eines Algorithmus ist darauf hinzuweisen, dass außerhalb der im Zusammenhang mit dieser Untersuchung und ihrer Zielsetzung betrachteten Aspekte weitere Stärken, aber auch potenzielle Risiken vorhanden sein können.

Obwohl die Durchführung der Analyse größtmöglicher Sorgfalt unterlag, schließt die TÜV Rheinland i sec GmbH daher jede Haftung für vorhandene und nicht erkannte potenzielle Risiken aus.

Das hier erarbeitete Ergebnis entbindet Spiele-Palast in keiner Weise von der Weiterverfolgung seiner Ziele. Das Unternehmen ist in jedem Fall für seine Maßnahmen zur Sicherstellung seiner Ziele selbst verantwortlich.

Jede Haftung für eventuelle Schäden, die aus einer falschen Anwendung der hier gegebenen Informationen resultieren, wird ausgeschlossen.