

# 28. BWINF, Runde 2, Aufgabe 2: Das Turmrestaurant

Matthias Springer, 28.0234.01

7. April 2010

## 1 Lösungsidee

Zunächst sollen nur die im Aufgabentext explizit geforderten Lösungsideen, also ein Programm ohne eigene Erweiterungen, vorgestellt werden. Das Turmrestaurant hat nur einen einzigen Tisch, der  $N_{Plätze}$  Sitzplätze enthält. Gruppen weisen eine Große von  $1 \leq k \leq k_{max}$  Personen auf, wobei der Algorithmus insgesamt mit maximal  $k_{max}$  Personen auskommen muss. Daraus ergibt sich ergibt sich für die Anzahl der Gruppen die Beschränkung  $1 \leq N_{Gruppen} \leq k_{max}$ . Es existiert außerdem ein Gruppenkoordinator (dieser zählt nicht zu den Personen), der das Gegenstück zum Ober darstellt und alle Gruppen-Entscheidungen trifft. Dazu gehören zum Beispiel das Erstellen von neuen Gruppen oder das Auflösen von bestehenden Gruppen. Wenn ein Tisch betrachtet wird, werden die einzelnen Plätze mit Indizes  $0 \leq i < N_{Plätze}$  durchnummeriert. Bei den meisten Algorithmen wird zunächst davon ausgegangen, dass der Tisch rechteckig ist. Erst später wird jeder Algorithmus so erweitert, dass er auch runde Tische verarbeiten kann. Das ist sinnvoll, weil ein Algorithmus dann viel einfacher erklärt werden kann und meist nur geringfügig angepasst werden muss, um auch runde Tische verarbeiten zu können.

Eine Lösungsstrategie umfasst immer einen Algorithmus für den Ober und den Gruppenkoordinator. Es ist möglich, verschiedene Algorithmen gegeneinander antretenzulassen.

### 1.1 Theoretische Vorüberlegungen

### 1.2 Einfaches Restaurant

Das einfache Restaurant besteht aus dem einfachen Ober und dem einfachen Gruppenkoordinator. Die Algorithmen tragen deshalb das Adjektiv *einfach*, weil außer dem *Divide-and-conquer*-Prinzip keine besonderen Strategien aus der Informatik verwendet werden. Es wurde lediglich darüber nachgedacht, wie ein schlauer Ober und ein schlauer Koordinator am besten handeln.

### 1.2.1 Einfacher Ober

Der einfache Ober verwendet ein ganz simples und triviales Verfahren, um möglichst viele Gruppen unterzubringen. Wenn der Tisch anfangs leer ist, platziert er die erste Gruppe an einer beliebigen Stelle. Wenn eine weitere Gruppe hinzugefügt werden soll, wird der Tisch auf freie Plätze untersucht. Die Gruppe wird dann am Anfang eines freien Platzintervalls platziert, das möglichst klein ist, aber genug Platz für die Gruppe bietet. Anschaulich bedeutet das, dass möglichst wenig Platz verschwendet werden soll. Eine Gruppe wird immer in der kleinstmöglichen Lücke platziert. Ein Algorithmus *Finde Subarray mit Mindestlänge* findet das optimale Intervall in linearer Zeit, indem einfach das ganze Array durchlaufen wird, wobei jederzeit gespeichert wird, wie lange das aktuelle Intervall gerade ist. Ein Intervall gilt dann als beendet, wenn bereits besetzte Plätze auf unbesetzte Plätze folgen. Wenn das gefundene Intervall mindestens so groß wie die Mindestlänge ist, wird es mit der besten, bereits gefundenen Lösung verglichen und ggf. gespeichert. Da es sich um einen kreisförmigen Tisch handelt, muss das Array in Wirklichkeit zweimal durchlaufen werden. Wenn der Algorithmus am Ende des Arrays angekommen ist, wird wieder am Anfang begonnen, als ob es sich um einen größeren, rechteckigen Tisch der Größe  $2N_{Plätze}$  handeln würde. Auf eine Besonderheit ist jedoch noch zu achten: Es muss zuerst eine geeignete Startposition gefunden werden, ab der das Array durchsucht wird. Würde man zum Beispiel immer am ersten Platz beginnen, könnte folgendes passieren.

#### Beispiel 1

		X	X			X	X	X	
--	--	---	---	--	--	---	---	---	--

(1)

Dieses Platzarray weist eine Lücke der Größe 3 und eine Lücke der Größe 2 auf. Beginnt man beim ersten Sitzplatz mit der Lückensuche, so würde der Algorithmus ganz am Anfang eine Lücke der Größe 2 finden und ganz zum Schluss (bei der zweiten Iteration über das Array) eine Lücke der Größe 1. Auf dieses Problem bin ich beim Debuggen durch Zufall gestoßen.

Man kann dieses Problem umgehen, indem man bei einem besetzten Sitzplatz mit der Suche beginnt. Dann startet der Algorithmus am Anfang einer Lücke und endet am Ende einer Lücke.

**Hinweis zu allen Beispielen** Jede Gruppe hat einen eigenen, eindeutigen Index. Wenn ein Tischplatzarray Zahlen ungleich Null enthält, sind damit Gruppen gemeint. Gleiche Zahlen deuten an, dass die Plätze von der gleichen Gruppe besetzt sind. Der Index 0 steht für einen unbesetzten Platz.

**Beispiel 2** Die folgenden Beispiele zeigen ein Tisch, dargestellt als eindimensionales Array. Jede Zahl steht dabei für eine Gruppe, der Buchstabe X bezeichnet einen bereits besetzten Platz. Anfangs habe der Tisch die folgende Belegung.

		X	X			X		X	
--	--	---	---	--	--	---	--	---	--

(2)

Wenn nun eine Gruppe der Größe 2 hinzugefügt werden soll, so gibt es nur eine erlaubte Stelle.

		X	X	1	1	X	X	
--	--	---	---	---	---	---	---	--

(3)

Die Gruppe darf nicht an einer anderen Stelle platziert werden, weil sonst Platz verschwendet werden würde. Man könnte sonst eine Gruppe der Größe 3 nicht mehr hinzufügen. Dies soll nun aber geschehen.

2	2	X	X	1	1	X	X	2
---	---	---	---	---	---	---	---	---

(4)

**Beispiel 3** Wenn es sich anfangs um einen leeren Tisch handelt, werden neue Gruppen von links nach rechts hinzugefügt. Werden beispielsweise eine Gruppe der Größe 2 und eine Gruppe der Größe 5 hinzugefügt, ergibt sich die folgende Belegung.

1	1	2	2	2	2	2		
---	---	---	---	---	---	---	--	--

(5)

Das Einfügen von neuen Gruppen hat beim einfachen Ober eine Laufzeitkomplexität von  $\mathcal{O}(n)$ . Wenn man von eigenen Erweiterungen absieht, gibt es keine weiteren Aktionen, die der einfache Ober ausführt. Er darf zum Beispiel keine Gruppen umsetzen, um den Platz effizienter zu nutzen, ebenso darf er keine Gruppen bevorzugen oder benachteiligen, indem er zum Beispiel kleine Gruppen ablehnt.

### 1.2.2 Einfacher Gruppenkoordinator

Während die Methode zum Platzieren neuer Gruppen beim einfachen Ober sehr einfach und offensichtlich war, kommt beim einfachen Gruppenkoordinator ein etwas schwierigerer Algorithmus zum Einsatz. Der einfache Koordinator versucht grundsätzlich, das Restaurant so weit wie möglich zu *zerstückeln*. Das gelingt ihm dadurch, dass er zunächst sehr viele kleine Gruppen<sup>1</sup> erstellt. Dann löscht er jede zweite Gruppe<sup>2</sup> und erstellt größere Gruppen, die in den entstandenen Intervallen keinen Platz finden. Das macht er so lange, bis das größte vorhandene Intervall kleiner als die Anzahl der freien Personen<sup>3</sup> ist. In einem solchen Fall hat der einfache Koordinator gewonnen, weil er dann nur eine große Gruppe erstellen muss, die der Ober unmöglich unterbringen kann.

Der einfache Koordinator arbeitet effizient, wenn er gegen einen einfachen Ober antritt. Effizient bedeutet in diesem Zusammenhang, mit möglichst wenigen freien Personen auszukommen<sup>4</sup>. Ein Ober, der die Strategie des einfachen Koordinators kennt, könnte natürlich versuchen, die Gruppen anders zu platzieren, indem er die Tatsache ausnutzt, dass immer jede zweite Gruppe aufgelöst wird und dann eine größere Gruppe erstellt wird. Aus diesem Grund wurde der Algorithmus so erweitert, dass er nicht notwendigerweise immer jede zweite Gruppe löscht. Der folgende Algorithmus funktioniert nur

<sup>1</sup>Dem einfachen Koordinator steht nur eine begrenzte Anzahl an freien Personen zur Verfügung. Diese kann er verschiedenen Gruppen zuteilen.

<sup>2</sup>Dabei werden die Personen in dieser Gruppe wieder zu freien Personen.

<sup>3</sup>Eine Person ist genau dann frei, wenn sie keiner Gruppe angehört.

<sup>4</sup>Natürlich wäre der einfache Koordinator noch besser, wenn er gegen einen richtig dummen Ober spielt.  
Der einfache Koordinator stellt das Gegenstück zum einfachen Ober dar.

unter der Annahme, dass der einfache Koordinator den Tisch und seine Belegung sehen kann<sup>5</sup>.

Der einfache Koordinator benötigt als Eingabe zunächst eine Folge von natürlichen Zahlen größer Null, wobei jede Zahl größer als ihr Vorgänger sein muss. Es muss also eine Folge  $f_n \in \mathbb{N}$  mit  $f_1 > 0$  und  $f_{n+1} > f_n$  vorliegen. Es werden dann  $n$  Runden durchgeführt. In einer Runde versucht der einfache Koordinator zunächst, möglichst viele bereits platzierte Personen zu entfernen<sup>6</sup>. Jedoch dürfen in der  $i$ -ten Runde dabei keine neuen Lücken (Intervalle) entstehen, die größer als  $f_i - 1$  sind. Aus den freien Personen werden dann möglichst viele Gruppen der Größe  $f_i$  gebildet<sup>7</sup>.

Anschaulich erklärt passiert also Folgendes. Es werden zunächst kleine Gruppen gebildet. Der einfache Ober ordnet diese *von links nach rechts* an. Dann werden einzelne Gruppen so gelöscht, dass Lücken (Intervalle) mit einer maximalen Größe von  $f_i - 1$  auftreten und gleichzeitig möglichst viele Personen frei werden. Anschließend bildet der einfache Koordinator aus den frei gewordenen Personen neue Gruppen der Größe  $f_i$ , die nicht in die entstandenen Lücken hineinpassen. So wird der Tisch mehr und mehr *fragmentiert* (zerstückelt). Bereits vor der Anwendung des Algorithmus bestehende Lücken, die zu groß sind (also  $f_i$  oder größer), dürfen nicht noch weiter vergrößert werden. Warum das so ist und eine genaue Beschreibung des Algorithmus befinden sich in einem extra Kapitel.

**Beispiel 4** Wenn es sich anfangs um einen leeren Tisch handelt und die Folge  $(1, 2)$  vorliegt, mit  $k_{max} = 8$ , ergibt sich die folgende Situation. Zunächst, werden 8 Gruppen der Größe  $f_1 = 1$  gebildet. Der einfache Ober würde diese von links nach rechts anordnen.

1	2	3	4	5	6	7	8		
---	---	---	---	---	---	---	---	--	--

(6)

Nun werden möglichst viele Gruppen gelöscht, sodass neue Lücken mit einer maximalen Größe von  $f_2 - 1 = 2 - 1 = 1$  entstehen. Dabei werden 3 Personen frei, eine bessere Lösung existiert scheinbar nicht<sup>8</sup>. Gruppe 8 darf nicht entfernt werden, weil die (schon zu große) Lücke am rechten Rand sonst vergrößert werden würde. Selbiges gilt für Gruppe 1<sup>9</sup>.

1		3		5		7	8		
---	--	---	--	---	--	---	---	--	--

(7)

Es stehen jetzt 3 freie Personen zur Verfügung, der nächste Schritt würde eigentlich darin bestehen, neue Gruppen der Größe 2 zu erstellen. Das ist jedoch notwendig, weil eine Gruppe der Größe 3 bereits ausreicht, um den Ober zu ärgern.

<sup>5</sup>Die Aufgabenstellung lässt das m.E. offen. Die restlichen Schüler (freie Personen) befinden sich schließlich außerhalb des Restaurants und können womöglich nicht durch die Fenster des sehr hohen Turmrestaurants schauen. Neben den Schülern befinden sich u.U. auch noch *normale* Gäste im Restaurant.

<sup>6</sup>Der Koordinator kann natürlich immer nur ganze Gruppen entfernen.

<sup>7</sup>Deshalb gilt  $f_n \leq k_{max}$ .

<sup>8</sup>Es folgt später ein Algorithmus, der entscheidet, welche Gruppen entfernt werden sollen, damit das Maximum (oder zumindest eine gute Näherung) an Personen frei wird.

<sup>9</sup>Man bedenke, dass der Tisch rund ist.

**Beispiel 5** Hier ein noch etwas aussagekräftigeres Beispiel. Eigentlich handelt es sich hierbei schon um einen Testfall, der vom Programm verarbeitet wurde. Alle Aktionen wurden vom fertigen Programm durchgeführt. Es gilt  $k_{max} = 11$  und  $N_{Plätze} = 20$ . Außerdem liegt die Folge  $(1, 2, 4, 8, 16)$  vor. Anfangs ist das Restaurant noch leer.

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

(8)

Jetzt werden 11 Gruppen mit je einer Person erstellt, weil  $f_1 = 1$ . Der einfache Ober platziert die Gruppen so, dass kein Platz verschwendet wird.

1	2	3	4	5	6	7	8	9	10	11								
---	---	---	---	---	---	---	---	---	----	----	--	--	--	--	--	--	--	--

(9)

Jetzt befindet sich der Koordinator im zweiten Zustand.  $f_2 = 2$ , deshalb versucht der Koordinator, so viele Gruppen wie möglich zu entfernen, ohne dass (neue) Lücken entstehen, die größer als 1 sind<sup>10</sup>.

1		3		5		7		9		11								
---	--	---	--	---	--	---	--	---	--	----	--	--	--	--	--	--	--	--

(10)

Jetzt stehen wieder 5 freie Personen (4 durch Auflösung der Gruppen, 1 noch von vorher) zur Verfügung. Die werden verwendet, um neue Gruppen der Größe  $f_2 = 2$  zu erstellen. Es werden also zwei Gruppen mit je zwei Personen erstellt (eine Person bleibt übrig). Die passen natürlich nicht in die gerade erstellten Lücken. Der einfache Ober platziert sie wieder möglichst platzsparend.

1		3		5		7		9		11	12	12	13	13				
---	--	---	--	---	--	---	--	---	--	----	----	----	----	----	--	--	--	--

(11)

Jetzt befindet sich der Koordinator im dritten Zustand.  $f_3 = 4$ , deshalb versucht der Koordinator wieder, Gruppen zu entfernen, wobei keine neuen Lücken größer als 3 entstehen dürfen.

1		3			7			11			13	13						
---	--	---	--	--	---	--	--	----	--	--	----	----	--	--	--	--	--	--

(12)

Jetzt stehen wieder 5 freie Personen zur Verfügung. Wegen  $f_3 = 4$  wird jetzt eine Gruppe der Größe 4 erstellt. Diese passt wieder nicht in die erstellten Lücken. Der Ober platziert die Gruppe wieder platzsparend.

1		3			7			11			13	13	14	14	14	14		
---	--	---	--	--	---	--	--	----	--	--	----	----	----	----	----	----	--	--

(13)

Jetzt befindet sich der Koordinator im vierten Zustand.  $f_4 = 8$ , deshalb versucht der Koordinator wieder, Gruppen zu entfernen, wobei keine neuen Lücken größer als 7 entstehen dürfen.

1					7					13	13							
---	--	--	--	--	---	--	--	--	--	----	----	--	--	--	--	--	--	--

(14)

Nun stehen dem Koordinator 7 freie Personen zur Verfügung, die größte Lücke ist aber nur 6 Plätze groß. Der Koordinator erstellt also eine Gruppe der Größe 7. Die hat keinen Platz mehr am Tisch. Der Ober ärgert sich.

Im Folgenden sollen die einzelnen Schritte des Algorithmus genauer erklärt werden.

<sup>10</sup>Später wird erklärt, wie genau der Algorithmus Gruppen entfernt.

### 1.2.3 Löschen bestehender Gruppen

Der erste Schritt des Koordinator-Algorithmus besteht darin, so viele Personen wie möglich zu entfernen, ohne dabei zu große Lücken entstehen zu lassen. Eine Lücke ist genau dann zu groß, wenn sie in der  $i$ -ten Runde  $f_i - 1$  überschreitet und vor dem Entfernen der Gruppe noch nicht zu groß war oder gar nicht existierte. Die Kunst liegt nun darin, zu entscheiden, welche Gruppen entfernt werden sollen. Ein trivialer Brute-Force-Algorithmus mit exponentieller Laufzeit könnte einfach alle Kombinationen ausprobieren und dann überprüfen, ob die Kombination gültig ist<sup>11</sup> und die Anzahl der entfernten Personen ermitteln. Ein solcher Algorithmus hätte jedoch exponentielle Laufzeit, weil  $2^{N_{\text{Gruppen}}}$  viele Situationen betrachtet werden müssen (ohne die trivialen Fälle zu streichen). Eine andere Möglichkeit wäre, eine Heuristik zu verwenden, die eine möglichst gute, womöglich aber nicht optimale Lösung, ermittelt. Dazu könnte vielleicht ein Greedy-Algorithmus entwickelt werden.

Ich habe einen *gierigen Divide-and-conquer*-Algorithmus, der das Problem in polynomieller Zeit lösen kann, entwickelt. Leider konnte ich nicht beweisen, dass dieser Algorithmus immer die optimale Lösung findet, eigene Tests weisen aber darauf hin. Der Algorithmus erinnert etwas an die Divide-and-Conquer-Lösung des *Maximum-Subarray Problems*<sup>12</sup>.

**Beispiel 6** Der Algorithmus soll auf diesen Tisch angewandt werden, mit  $f_i = 4$ , d.h. es dürfen keine neuen Lücken entstehen, die größer als 3 sind.

1		3		5		7		9		11	12	12	13	13				
---	--	---	--	---	--	---	--	---	--	----	----	----	----	----	--	--	--	--

(15)

Zuerst teilt man den Tisch in zwei Hälften, und zwar möglichst in der Mitte, weil der Algorithmus dann hoffentlich nicht so viele rekursive Aufrufe machen muss<sup>13</sup>. Dazu muss aber zuerst eine geeignete Stelle zur Trennung gefunden werden. Eine Stelle ist genau dann *nicht* geeignet, wenn durch die Trennung eine Gruppe *auseinander gerissen* wird. Eine gute Stelle kann man finden, indem man sich von der Mitte aus so lange nach links oder nach rechts bewegt, bis eine neue Gruppe beginnt oder ein leerer Platz gefunden wird. Das *Divide-and-conquer*-Prinzip schlägt vor, nun beide Teile rekursiv zu bearbeiten, wobei die Problemgröße jedes Mal reduziert wird.

**Beispiel 7** Eine Teilung findet also wie folgt statt. Es wird keine Gruppe getrennt.

1		3		5		7		9	
---	--	---	--	---	--	---	--	---	--

(16)

11	12	12	13	13				
----	----	----	----	----	--	--	--	--

(17)

Hierbei gibt es nun vier Fälle, die betrachtet werden müssen. Es muss entschieden werden, ob die Gruppen am rechten Rand des linken Teiles (Gruppe A) und am linken Rand des rechten Teiles (Gruppe B) entfernt werden sollen oder nicht.

<sup>11</sup>D. h. ob nun zu große Lücken vorliegen oder nicht.

<sup>12</sup>die mich nebenbei bemerkt auf diesen Algorithmus gebracht hat.

<sup>13</sup>Vgl. Quicksort: Nur wenn die Trennung immer in der Mitte stattfindet arbeitet der Algorithmus schnell.

**Beispiel 8** Gruppe A ist in diesem Fall die Gruppe mit dem Index 9 und Gruppe B ist die Gruppe mit dem Index 11.

Diese beiden Entscheidungen können nicht unabhängig voneinander getroffen werden, weil das Entfernen beider Gruppen vielleicht eine zu große Lücke erzeugen könnte. Um eine gute und gültige Entscheidung an den Rändern treffen zu können, muss der Algorithmus darüber Bescheid wissen, was auf der anderen Seite des Randes geschehen ist. Deshalb werden die vier Fälle *Gruppe A und Gruppe B entfernen*, *Nur Gruppe A entfernen*, *Nur Gruppe B entfernen* und *Keine Gruppe entfernen* ausgeführt. Wenn eine Gruppe nicht entfernt wird, so wird sie als *fest* markiert, das heißt, sie darf überhaupt nicht entfernt werden, auch nicht im nächsten rekursiven Aufruf. Für jeden der vier Fälle wird zunächst die Gültigkeit überprüft, das heißt, ob durch Entfernen einer Gruppe (A und/oder B) bereits eine zu große Lücke entsteht. Wenn dies zutrifft, wird der Fall gar nicht weiter betrachtet. Ansonsten werden zwei rekursive Aufrufe, für den linken und den rechten Teil jeweils einen, gestartet.

**Beispiel 9** An dieser Stelle müsste man nun normalerweise alle vier Fälle betrachten. Darauf verzichte ich und wähle nur die beste Entscheidung. Natürlich ist anfangs noch nicht klar, welche Entscheidung die Beste ist.

1	3	5	7			
---	---	---	---	--	--	--

(18)

-11	12	12	13	13				
-----	----	----	----	----	--	--	--	--

(19)

Gruppe A wurde entfernt und Gruppe B wurde als *fest* markiert, was damit angedeutet werden soll, dass Gruppe B nun einen negativen Index hat. Das Sitzplatzarray bleibt gültig, weil keine neue Lücke erzeugt wurde, die größer als 3 ist.

Ein Aufruf des Algorithmus führt also normalerweise zu  $2 \cdot 4 = 8$  rekursiven Aufrufen<sup>14</sup>. Wenn linker und rechter Teil rekursiv bearbeitet wurden, werden beide Teile wieder zusammengesetzt und auf Gültigkeit überprüft. Dies geschieht für alle vier Fälle und der beste Fall wird gespeichert und zurückgegeben (in das Platzarray zurückgeschrieben<sup>15</sup>). Ein Fall ist dann gut, wenn er möglichst viele Personen freigemacht hat und keine zu große Lücke entstanden ist (Gültigkeit).

**Beispiel 10** Die beiden Subarrays (beide Teile des Platzarrays) werden nun rekursiv verarbeitet, mit dem gleichen Algorithmus. Diese Schritte überspringe ich hier bewusst. Zum Schluss sehen die beiden Subarrays wie folgt aus.

1	3			7		
---	---	--	--	---	--	--

(20)

11		13	13				
----	--	----	----	--	--	--	--

(21)

<sup>14</sup>Eigentlich würden insgesamt 4 Aufrufe genügen, später mehr dazu.

<sup>15</sup>Es handelt sich also um einen *Out-of-place-Algorithmus*, der zusätzlichen Platz benötigt. Ein *In-place-Algorithmus* bietet sich nicht an, weil vier Fälle beachtet werden müssen.

Die beiden Subarrays werden wieder zusammengesetzt.

1	3			7			11			13	13				
---	---	--	--	---	--	--	----	--	--	----	----	--	--	--	--

(22)

Das Sitzplatzarray ist noch immer gültig, weil keine neue Lücke erstellt wurde, die größer als 3 ist. Die 5 freien Plätze am Ende des Arrays waren auch vor Aufruf des Algorithmus schon frei.

Nun gilt es noch Abbruchbedingungen für die Rekursion zu finden. Wenn ein Tisch (d.h. ein Subarray) keine Gruppe mehr enthält, die entfernt werden kann, kann abgebrochen werden<sup>16</sup>.

#### 1.2.4 Spezialfälle und Besonderheiten des Algorithmus

Besonderheiten ergeben sich hauptsächlich aus der Tatsache, dass der Tisch im Restaurant rund ist. Wie der Algorithmus genau angepasst wurde, wird in der Programm-Dokumentation beschrieben, es sei jedoch gesagt, dass es in den meisten Fällen genügt, einfach zweimal (anstatt nur einmal) über das Platzarray zu iterieren.

Ich konnte zwar nicht beweisen, dass der Algorithmus immer eine optimale Lösung findet, jedoch terminiert er mit Sicherheit immer mit einem gültigen Ergebnis. Zu Problemen kann es nur dadurch kommen, dass beim Entfernen der Gruppen A/B eine zu große Lücke entsteht. An dieser Stelle kann zuvor noch keine zu große Lücke gewesen sein, weil der Algorithmus an dieser Stelle noch keine Änderung(en) vorgenommen hat. Diese entstehen nur durch rekursive Aufrufe und immer an den Rändern des Platzarrays. Es ist zwar möglich, dass zum Beispiel beim Entfernen von Gruppe A *und* Gruppe B eine zu große Lücke entsteht, jedoch wird dieser Fall dann verworfen. Es gibt immer auch einen Fall, der keine Änderung durchführt, nämlich wenn Gruppe A und Gruppe B als *fest* markiert werden. Dann bleibt das Platzarray auf jeden Fall gültig. Da es sich um einen runden Tisch handelt, gibt es zudem noch die Möglichkeit, dass an den Rändern des (noch nicht geteilten) Platzarrays eine zu große Lücke entsteht, wenn zum Beispiel die beiden äußersten Gruppen entfernt werden. Deshalb generiert der Algorithmus ganz zu Anfang nochmals vier Fälle, analog zur bereits erklärten Fallunterscheidung. Nur geht es dieses Mal um die Gruppe ganz links und die Gruppe ganz rechts. Auch an den Rändern kann es deshalb nicht zu Problemen kommen, weil es immer mindestens einen gültigen Fall gibt, nämlich den Fall, in dem die beiden äußersten Gruppen als *fest* markiert werden.

Es wurde im vorherigen Kapitel behauptet, dass bestehende, bereits zu große Lücken, nicht weiter vergrößert werden sollen. Das hat den Hintergrund, dass der *einfache Koordinator* möglichst effizient gegen den *einfachen Ober* vorgehen soll. Wenn man nun eine bereits zu große Lücke noch weiter vergroßern würde, würde der *einfache Ober* den/die frei gewordenen Platz/Plätze (am Rand einer großen Lücke) wahrscheinlich gleich wieder besetzen, jedoch mit einer größeren Gruppe. Dadurch wird das Problem nur schwieriger für den einfachen Koordinator, weil er nun nicht mehr so viele Möglichkeiten hat, Gruppen zu entfernen.

<sup>16</sup>Achtung: Eine Gruppe kann auch dann nicht entfernt werden, wenn sie als *fest* markiert wurde.

**Beispiel 11** Ein Tisch der Größe 10 habe zur Zeit 6 Gruppen. Der einfache Koordinator soll nun Gruppen entfernen, wobei die maximale Lückengröße bei 1 liegt ( $f_i = 2$ ).

1	2	3	4	5	6			
---	---	---	---	---	---	--	--	--

(23)

Der Koordinator entfernt zwei Gruppen, sodass sich die folgende Situation ergibt.

1		3		5	6			
---	--	---	--	---	---	--	--	--

(24)

Die Gruppe mit dem Index 6 darf nicht entfernt werden, weil sonst eine bereits zu große Lücke vergrößert werden würde. Angenommen, der Koordinator macht das trotzdem und platziert dann eine neue Gruppe der Größe 2, dann ergibt sich die folgende Situation.

1		3		5	7	7		
---	--	---	--	---	---	---	--	--

(25)

Der einfache Ober hat die neue Gruppe an der Position platziert, an der sich vorher die Gruppe mit dem Index 6 befand. Meiner Meinung nach wäre es sinnvoller gewesen, die Gruppe mit dem Index 6 nicht zu entfernen, sodass sich die folgende Situation ergibt.

1		3		5	6	7	7	
---	--	---	--	---	---	---	---	--

(26)

Im nächsten Schritt hat der Algorithmus dann nämlich mehr Möglichkeiten, Gruppen zu entfernen. Außerdem sind so keine freien Personen mehr übrig, zuvor war eine Person übrig geblieben.

Die Aufgabenstellung wurde von Anfang an so erweitert, dass auch *normale* Personen das Restaurant besuchen können. Intern werden solche Personen (es kann sich auch um Gruppen handeln) wie *feste* Personen (Gruppen) behandelt.

### 1.2.5 Optional: Laufzeitanalyse für das Löschen bestehender Gruppen

Die Laufzeitkomplexität für den zuvor vorgestellten Algorithmus soll nun grob untersucht werden, damit festgestellt werden kann, inwiefern dieser Algorithmus eine Verbesserung im Gegensatz zum Brute-Force-Algorithmus darstellt. Dazu soll die *Master-Methode für das Lösen von Rekurrenzen*<sup>17</sup> verwendet werden. Die Master-Methode funktioniert für Rekurrenzen der Form  $T(n) = aT(n/b) + f(n)$ . Weil der Tisch in zwei Teile geteilt wird und acht rekursive Aufrufe erfolgen, sind  $a = 8$  und  $b = 2$ . Die Funktion  $f(n)$  ist auf jeden Fall linear, weil für das Prüfen, Zusammenfügen und Berechnen der Güte einer Lösung jeweils maximal linearer Zeitaufwand<sup>18</sup> notwendig ist. Somit ist  $f(n) \in \mathcal{O}(n)$  und der erste Fall der Master-Methode kann angewendet werden, da  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) = \mathcal{O}(n^{3-\epsilon}) = \mathcal{O}(n)$  mit  $\epsilon = 2$ . Daher gilt  $T(n) = \theta(n^{\log_b a}) = \theta(n^3)$ . Das Löschen bestehender Gruppen kann also in polynomieller Zeit durchgeführt werden, was eine gute Verbesserung im Vergleich zum exponentiellen Ansatz darstellt. In Wirklichkeit könnte der Algorithmus sogar in  $\theta(n^2)$  implementiert werden, wenn nur

<sup>17</sup>Nach Cormen: Introduction to Algorithms

<sup>18</sup>Siehe Programm-Dokumentation

vier rekursive Aufrufe gemacht werden<sup>19</sup>. Da manche Fälle schon frühzeitig als ungültig erkannt werden und dadurch rekursive Aufrufe ausbleiben, gilt die angegebene Laufzeit eigentlich nur als obere Schranke.

## 1.3 Eigene Erweiterungen

### 1.3.1 Der perfekte Ober

Wie kann man feststellen, ob die entwickelten Algorithmen gut sind oder nicht? Die wohl sicherste Methode würde einfach alle Spielzüge ausprobieren (*Brute-Force-Verfahren*). Leider gibt es so viele mögliche Spielzüge, dass das zeitlich kaum schaffbar ist. Geht man von einem Restaurant mit  $n$  Sitzplätzen und (der Einfachheit halber)  $n$  freien Personen aus, so gibt es allein im ersten Spielzug für den Koordinator  $2^n$  Möglichkeiten, eine (einige) Gruppe zusammenzustellen. Maximal kann er  $n$  Gruppen erstellen. Diese Zahl verringert sich etwas, da es nicht notwendig ist, zwischen den einzelnen Personen zu unterscheiden. Es ist lediglich notwendig, zu wissen, aus *wie vielen* Personen die einzelnen Gruppen bestehen. Trotzdem gibt es noch viel zu viele Spielzüge. Anfangs dachte ich an einen *Minimax- oder Alpha-Beta-Algorithmus*, jedoch habe ich schnell bemerkt, dass das wohl nicht machbar ist.

Die Anzahl der Spielzüge lässt sich drastisch reduzieren, wenn nur alle Spielzüge für den Ober generiert werden und für den Koordinator immer der Spielzug ausgeführt wird, den der einfache Koordinator machen würde. So kann man untersuchen, wie gut der einfache Koordinator arbeitet, weil er immer mit dem bestmöglichen Spielzug des Obers konfrontiert wird. Ebenso lassen sich Aussagen über die Güte des Algorithmus des einfachen Obers machen, wenn man untersucht, mit wie vielen Personen der einfache Ober im Gegensatz zum perfekten Ober zurechtkommt. Solche Aussagen sind jedoch mit Vorsicht zu genießen, später mehr dazu.

**Algorithmus des perfekten Obers** Damit der perfekte Ober immer die perfekte Lösung finden kann, muss er im *Brute-Force-Verfahren* alle möglichen Spielzüge, die ihm zur Verfügung stehen, überprüfen und die Reaktion des Gegners berechnen (und dann wieder alle möglichen Spielzüge für den Ober). Dies macht er rekursiv so lange, bis ein Spieler gewonnen hat oder ein Spieler aufgibt. Der Koordinator hat gewonnen, wenn der Ober sich ärgert, während der Ober gewonnen hat, wenn der Koordinator aufgibt (siehe Kapitel *Einfacher Koordinator*). Es ist wichtig, dass der Ober, nachdem er alle seine Spielzüge überprüft hat, immer den Spielzug wählt, bei dem er sicher gewonnen hat. Gibt es keinen solchen Spielzug, so hat er verloren. In Wirklichkeit ist es gar nicht notwendig, alle Spielzüge zu überprüfen. Wenn von einem Spielzug bekannt ist, dass er zum Sieg führt, brauchen die restlichen Spielzüge für den Ober gar nicht mehr überprüft werden<sup>20</sup>. Das Ergebnis eines rekursiven Aufrufs wird von der oberen Instanz (die Pro-

---

<sup>19</sup>Wenn die Ergebnisse der rekursiven Aufrufe gespeichert werden, kann man sich die Hälfte der Aufrufe sparen, weil jeder Aufruf doppelt geschieht. In C# ist das aber nicht so schön zu implementieren.

<sup>20</sup>Im *Alpha-Beta-Algorithmus* werden ganze Zweige des Rekursionsbaumes/Spielbaumes auf diese Art und Weise abgeschnitten (*pruned*).

zedur, die den rekursiven Aufruf ausgelöst hat) ausgewertet. Der Aufruf, der zum Sieg führt, wird ausgewählt. Zum Schluss ist bekannt, ob der Ober in der Ausgangssituation (der Tisch, Anzahl der freien Personen, besetzte Plätze am Tisch bekannt) bereits gewonnen hat oder nicht.

**Generieren aller möglichen Spielzüge** Da es sich um sehr viele mögliche Spielzüge handelt, sollte man, soweit möglich, versuchen, ungültige Züge sehr früh auszusortieren. Ebenso wichtig ist es, gleiche Berechnungen nicht mehrfach durchzuführen. Normalerweise würde man dazu eine Hashtabelle verwenden. Wenn man jedoch etwas länger über das Problem nachdenkt, kann man sich das sparen. Der Algorithmus wurde so entworfen, dass er grundsätzlich keine doppelten/gleichen Spielzüge berechnet. Der Informationsgehalt eines Spielzustandes wurde möglichst klein gehalten.

Jeder Spielzustand enthält neben einem Sitzplatzarray, die aktuelle Rekursionstiefe, das ist notwendig, weil der einfache Koordinator abhängig von der Rekursionstiefe unterschiedliche Entscheidungen treffen kann. *Mit der Rekursionstiefe nehmen auch interner Zustand und maximale Lückengröße zu..*

Zum eigentlichen Generieren aller Spielzüge wurde eine externe Bibliothek verwendet. Die Menge  $S$  enthalte die Indizes aller freien Plätze am Tisch. Sollen für  $k$  zu platzierende Gruppen alle Spielzüge generiert werden, so werden alle Möglichkeiten generiert,  $k$  Elemente aus  $S$  ohne Beachtung der Reihenfolge (also ohne Permutationen) und ohne Wiederholungen zu entnehmen.

**Beispiel 12** In diesem Beispiel liegt der folgende Tisch der Größe 10 vor.

$$\boxed{1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid \mid \mid 7} \quad (27)$$

Dann wird die folgende Menge  $S$  generiert, die die Indizes aller leeren Plätze enthält.

$$S = 4, 8, 9 \quad (28)$$

Mit  $k = 2$  zu platzierenden Gruppen werden die folgenden Mengen generiert.

$$\{\{4, 8\}, \{4, 9\}, \{8, 9\}\} \quad (29)$$

Die folgenden Schritte werden für alle generierten Mengen ausgeführt. Es liegt nun eine Menge von  $k$  Zahlen vor, wobei  $s_i$  (Startposition) die  $i$ -te Zahl bezeichne. Die zu platzierenden Gruppen befinden sich in einer Liste, weisen also eine feste Reihenfolge auf. Die  $i$ -te Gruppe wird nun im Platzintervall  $[s_i; s_i + \text{Gruppengröße}]$  platziert. Jetzt muss noch überprüft werden, ob dieser Fall überhaupt gültig ist. Es könnte beispielsweise sein, dass Gruppen sich überschneiden oder bereits besetzte Sitzplätze beanspruchen. Dazu wird ein boolsches Array des Tisches angelegt, das immer dann auf *wahr* steht, wenn ein Platz schon besetzt ist. Nun werden die neuen Gruppen im Array eingetragen und die entsprechenden Arrayslots auf *wahr* gestellt. Ein Spielzug ist genau dann nicht gültig, wenn ein Arrayslot auf *wahr* gesetzt werden soll, der bereits auf *wahr* steht, also wenn ein bereits vergebener Sitzplatz nochmals besetzt werden soll.

**Beispiel 13** Will man eine Gruppe der Größe 2 am Tisch des obigen Beispiels positionieren, scheidet Position [4; 6] aus, weil Platz 5 bereits von der Gruppe mit dem Index 4 besetzt ist.

Als die  $k$  Indizes aus der Liste der Indizes aller freien Plätze ausgewählt wurden, habe ich ganz bewusst darauf geachtet, dass die einzelnen Permutationen nicht beachtet werden. Das liegt daran, dass beim einfachen Koordinator alle Gruppen die gleiche Größe aufweisen (wenn man die Option *Gruppen auffüllen* deaktiviert). Deshalb ist es egal, an welcher Stelle welche Gruppe beginnt. Gruppen mit gleicher Größe werden nicht unterschieden und weil alle Gruppen hier die gleiche Größe aufweisen, geht es nur um die Startpositionen.

Auf diese Art und Weise sollte kein Fall doppelt generiert werden, ebenso werden alle möglichen Fälle abgedeckt.

**Laufzeitanalyse für den perfekten Ober** Da es sich um einen *Brute-Force-Algorithmus* handelt, braucht man kein besonders gutes Laufzeitverhalten erwarten. Für größere Fälle ist er praktisch nicht anwendbar. Wenn  $k$  Startpositionen aus  $n$  möglichen Plätzen ausgewählt werden sollen, so gibt es allein dafür  $\binom{n}{k}$  Möglichkeiten. Ich habe mich (hauptsächlich auf Grund von Zeitmangel) nicht weiter damit beschäftigt, den Algorithmus zu beschleunigen.

### 1.3.2 Reservierungen im Turmrestaurant

Der Ober hat es nun endgültig satt, jeden Abend von einer Horde kleiner Schüler überlaufen zu werden, während seine eigentlichen Gäste, die in der Regel viel mehr Geld da lassen, keinen Platz mehr bekommen. Deshalb können seit letzter Woche Plätze reserviert werden. Innerhalb kürzester Zeit ist das Turmrestaurant jedoch so bekannt und beliebt geworden, dass sich der Ober eines Computersystems behelfen muss, um die viel zu vielen Reservierungswünsche bearbeiten zu können. Das Computersystem soll aus einer Auswahl von  $n$  Reservierungswünschen diejenigen auswählen, die am profitabelsten sind. Jeder Gast gibt bei der Reservierung an, wie viele Plätze er benötigt und wie viel er für die Reservierung bezahlen will. Es dürfen natürlich nicht so viele Reservierungen angenommen werden, dass mehr Plätze benötigt werden, als überhaupt verfügbar sind. Da das Turmrestaurant immer nur spät abends, kurz vor Ladenschluss so überfüllt ist, kann pro Tag nur zu einer einzigen Uhrzeit reserviert werden und danach schließt das Restaurant.

Die Schwierigkeit liegt bei dieser Entscheidung darin, zu entscheiden, welche Gruppen zugelassen werden sollen. Theoretisch gäbe es  $2^n$  Möglichkeiten (ohne Ausschluss der trivialen Fälle), die man mit einem *Brute-Force-Algorithmus* durchprobieren kann. Oder man verwendet *dynamische Programmierung* und löst das Problem (ebenfalls optimal) mit einer Laufzeitkomplexität von  $\mathcal{O}(n \cdot \text{Restaurantgröße})^{21}$ .

---

<sup>21</sup>Ich persönlich würde letztere Variante bevorzugen.

**Definition 1** Eine 2D-Matrix  $P[s][g]$  enthalte für jeden Wert von  $1 \leq g \leq g_{max}$  und  $1 \leq s \leq \text{Restaurantgröße}$  den maximal erzielbaren Profit, wobei die Basisfälle  $P[0][0] = P[0][1] = \dots = P[0][g_{max}] = 0$  und  $P[0][0] = P[1][0] = \dots = P[\text{Restaurantgröße}][0] = 0$  gelten.

Jede Reservierung hat einen eindeutigen Index und  $g_{max}$  bezeichnet die Anzahl der eingegangen Reservierungen. Die Variable  $s$  gibt die Größe des Restaurants (Anzahl der Sitzplätze) im aktuellen Teilproblem an. Als Profit wird der Gewinn<sup>22</sup>, also die Summe aller Einnahmen durch Reservierungen bezeichnet. Wenn das Restaurant keine Sitzplätze hat  $s = 0$ , kann das Restaurant keinen Profit machen, weil keine Reservierungen angenommen werden können. Analog kann es keinen Profit geben, wenn es keine Reservierungen gibt  $g = 0$ . Aus diesem Grund gelten die definierten Basisfälle.

Mit dynamischer Programmierung kann nun die optimale Lösung gefunden werden, indem die Matrix zeilenweise gefüllt wird. Der Algorithmus basiert auf dem *0-1-Knapsack-Problem*, bei dem es gilt, aus einer Menge  $S$  diejenigen Gegenstände (oder was auch immer) auszuwählen, die den Gewinn maximieren (Variationen sind möglich), wobei gleichzeitig ein bestimmtes Gesamtgewicht nicht überschritten werden darf.

**Definition 2** Die optimale Lösung, also der maximale Profit von  $P[s][g]$  kann über DP berechnet werden.  $P[s][g] = \max(P[s][g-1]; P[s-w(g)][g-1] + \text{prof}(g))$ , wobei  $w(g)$  die Größe der Gruppe  $g$  bezeichnet und  $\text{prof}(g)$  der Profit ist, den die Gruppe  $g$  verursacht.

Wie aus der Formel für  $P[s][g]$  ersichtlich ist, gibt es zwei Entscheidungsmöglichkeiten (der *max*-Ausdruck). Entweder man fügt die neue Reservierung  $g$ <sup>23</sup> hinzu oder nicht. Falls man die Reservierung nicht übernimmt, befindet man sich im Fall  $P[s][g-1]$ . Die Restaurantgröße ist gleich, aber die Reservierung  $g$  stand noch nicht zur Verfügung. Falls man die Reservierung übernimmt, kann man auf den Wert  $P[s-w(g)][g-1]$  zurückgreifen. Die Restaurantgröße wird um die aktuelle Reservierungsgröße verringert und die aktuelle Reservierung gab es noch nicht. Dann wird der Profit der aktuellen Reservierung  $g$  aufaddiert. Der Algorithmus fügt also erst einmal die aktuelle Reservierung hinzu und schlägt dann in der Matrix nach, welchen Profit das Restproblem (verringerte Restaurantgröße und aktuelle Reservierung nicht verfügbar) verursacht. Der gesuchte Wert ist in der Matrix schon verfügbar, da alle Fälle mit einer kleineren Anzahl an Reservierungen als  $g$  schon berechnet wurden. Genau deshalb ist dieser Algorithmus auch so effizient<sup>24</sup>.

Der maximale Profit des Problems kann aus der Matrix-Zelle  $P[\text{Restaurantgröße}][g_{max}]$  ausgelesen werden. Um die einzelnen Schritte rekonstruieren zu können, ist es jedoch

<sup>22</sup>Stimmt eigentlich nicht, der Gewinn ergibt sich aus *Einnahmen - Ausgaben*. Ausgaben sollen hier aber nicht betrachtet werden.

<sup>23</sup>Jede Reservierung hat einen eindeutigen Index  $g$  im Bereich von 1 bis  $g_{max}$ . In den einzelnen Fällen  $P[s][1 \leq g \leq g_{max}]$  kommen mit zunehmendem  $g$  immer mehr Reservierungen hinzu.

<sup>24</sup>Würde man gleich auf den Finalwert  $P[\text{Restaurantgröße}][g_{max}]$  zugreifen und rekursive Aufrufe verwenden, würden viele Werte mehrfach berechnet werden und man erhielte wahrscheinlich keine polynomiale Laufzeit.

noch notwendig, eine zweite Matrix zu benutzen, die die einzelnen Entscheidungen speichert, damit der *Weg* in der Matrix zurückverfolgt werden kann. Es muss jede Entscheidung gespeichert werden, also ob die Reservierung übernommen wurde oder nicht. Zum Schluss kann man die Einzelentscheidungen von der Matrix-Zelle  $P[Restaurantgröße][g_{max}]$  aus zurückverfolgen.

Die Aufgabenstellungen weist darauf hin, noch ein paar Sätze darüber zu verlieren, weshalb die Erweiterung *sinnvoll ist*<sup>25</sup>. Eine kleine Erklärung habe ich bereits zu Beginn des Kapitels abgegeben. Man könnte nun noch versuchen, einen Reservierungsplan für einen ganzen Tag zu erstellen, wobei darauf zu achten ist, dass sich Reservierungen nicht überschneiden. Das wäre aber dann wahrscheinlich zu aufwendig für eine Erweiterung. Natürlich ist es in der Realität nicht so, dass alle Reservierungen zur gleichen Zeit stattfinden. Die Schwierigkeit, die Personen zu platzieren fällt ebenfalls weg. Es ist in diesem Fall trivial, die Gruppen so zu setzen, dass Gruppenmitglieder immer geschlossen nebeneinander sitzen.

### 1.3.3 Kontostand für einzelne Schüler

Der Koordinator schickt die Schüler meistens mehrmals nacheinander in das Restaurant. Bisher wurde nicht beachtet, dass das auf keinen Fall kostenlos ist. Jede Person, die das Restaurant betritt, muss etwas essen oder trinken und kann nicht nur einen Platz blockieren. Deshalb soll jeder Besuch im Restaurant für jeden Schüler eine Geldeinheit kosten. Jeder Schüler hat einen Kontostand und bei jedem Besuch des Restaurants wird eine Geldeinheit abgezogen. Wenn der Kontostand bei Null angekommen ist, hat der Schüler kein Geld mehr und kann das Restaurant nicht mehr besuchen.

Wenn einige Schüler nur einen geringen Kontostand aufweisen und relativ viele Schüler zur Verfügung stehen, soll es auch möglich sein, dass ein Schüler nur jedes zweite oder dritte Mal in eine Gruppe eingebunden wird. Ein Algorithmus soll die Schüler so auswählen, dass jeder Schüler möglichst den gleichen Kontostand aufweist. Jeder Schüler kann so angeben, wie viel (Geld) es ihm wert ist, dass der Ober sich ärgert. Dazu wird ein *Heap* verwendet. Ein Heap ist eine Datenstruktur, die Elemente aufnehmen kann und Elemente sortiert zurückgibt. Genau genommen wird ein *Max-Heap* verwendet, der immer das größte Element zurückgibt. Ein Schüler ist verglichen mit einem anderen Schüler genau dann größer, wenn er einen größeren Kontostand aufweist. Anstatt eines Heaps könnte man auch eine sortierte Liste verwenden. Das Sortieren einer Liste hat eine Laufzeitkomplexität von  $\mathcal{O}(n \log n)$ . Das Einfügen von  $n$  Elementen in einen Heap weist die gleiche Laufzeitkomplexität auf ( $\mathcal{O}(\log n)$  pro Person). Ein Heap lohnt sich also nur dann, wenn nie alle Elemente aus dem Heap entnommen werden oder immer nur einzelne Elemente wieder in den Heap eingefügt werden. Beim Sortieren einer Liste werden nämlich immer alle Elemente neu sortiert, während beim Heap nur die neuen Elemente beachtet werden<sup>26</sup>. Ein Heap lohnt sich also, wenn nie alle Personen aus dem

---

<sup>25</sup>Wie sinnvoll ist es, ein Restaurant so lange zu betreten und wieder zu verlassen, bis der Ober sich ärgert?

<sup>26</sup>Wobei sich eine Sortierfunktion auch so implementieren liese, dass bereits vorsortierte Passagen zu einer Laufzeitverbesserung führen.

Heap entnommen werden, um Gruppen zu bilden.

Anstatt eine Liste der freien Personen zu speichern, wird ein Heap verwendet. Wenn ein Schüler kein Geld mehr hat, wird er gelöscht. Die Schwierigkeit bei dieser Erweiterung lag darin, einen eigenen Heap zu implementieren<sup>27</sup> und die bestehenden Klassen so anzupassen, dass möglichst wenig Code erforderlich ist. Die bestehenden Klassen wurden so angepasst, dass die neue Koordinator-Klasse von der alten *Einfacher Koordinator*-Klasse erbt und nur einige wenige Funktionen überschreiben muss (Objektorientierte Programmierung).

**Wann funktioniert diese Erweiterung?** Normalerweise werden immer alle zur Verfügung stehenden freien Personen verwendet, wenn neue Gruppen gebildet werden. Dann nützt es nichts, einige Personen bevorzugt auszuwählen, weil alle Personen ausgewählt werden. Bei großen Tischgrößen und wenn die Option *Gruppen auffüllen* deaktiviert ist, ist das aber nicht immer so. Oftmals bleiben einzelne Personen übrig, weil es nicht genügend Personen sind, um eine weitere neue Gruppe zu erstellen. Das kann auch mehrmals hintereinander passieren. Es sollen die Personen übrig bleiben, die einen niedrigen Kontostand aufweisen.

Um den Heap zu testen, könnte man auch Personen ausblenden. Wenn zum Beispiel von 20 freien Personen 5 Personen ausgeblendet werden, stehen effektiv nur 15 Personen zur Verfügung. Es werden nicht immer die gleichen 5 Personen ausgeblendet, sondern die 5 Personen, die gerade den niedrigsten Kontostand haben. In Wirklichkeit macht das wenig Sinn, weil man die 5 Personen gleich beim Zerstückeln des Platzarrays verwenden könnte. Dann braucht man u.U. sogar weniger Durchläufe<sup>28</sup>, weil das Zerstückeln schneller geht.

### 1.3.4 Zuständigkeit einzelner Kellner

**Hinweis** Wenn in diesem Kapitel (gilt auch für die Programm-Dokumentation) der Begriff *Tischgröße* auftaucht, ist eigentlich der Begriff *Anzahl der Gruppen* gemeint.

Ein großes Turmrestaurant mit einem großen Tisch (z.B. 100 Plätze) ist zu groß für einen Ober. Er kann vielleicht noch die Aufgabe allein übernehmen, Gruppen zu platzieren, jedoch kann er unmöglich 100 oder mehr Personen alleine bedienen. Stattdessen gibt es eine Reihe an Kellnern, die das übernehmen. Es gilt nun zu entscheiden, welcher Kellner für welche Personen zuständig ist. Ein Kellner soll dabei immer nur für ganze Gruppen zuständig sein, weil in der Regel die Personen einer Gruppe geschlossen eine Bestellung aufgeben und geschlossen bezahlen. Außerdem sollen die Gruppen so unter den Kellnern aufgeteilt werden, dass jeder Kellner in etwa gleich viel Arbeit hat, d.h. dass jeder Kellner in etwa für gleich viele Personen zuständig ist. Damit ein Kellner nicht immer von einem Ende des Tisches zum Anderen laufen muss, soll ein Kellner immer für ein geschlossenes Tischsegment (Kreissegment) zuständig sein.

Wenn das Restaurant unter  $n$  Kellnern aufgeteilt werden soll, gilt es  $n$  Trennungen

---

<sup>27</sup>Das .NET Framework hat keine eigene Heap- oder PriorityQueue-Klasse.

<sup>28</sup>In einem Durchlauf werden Gruppen entfernt und neue Gruppen platziert.

des Tisches vorzunehmen, wobei ein Kellner immer für die Personen von einer Trennung bis zur Nächsten zuständig ist. Wenn es sich um einen nicht-kreisförmigen Tisch handelt, entspricht dieses Problem dem *Partitioning Problem* (wobei aber nur  $n - 1$  Trennungen vorgenommen werden müssen). Wenn man eine Lösung für das Partitioning Problem gefunden hat, kann der Algorithmus aber so angepasst werden, dass er auch für kreisförmige Tische funktioniert.

**Lösungsidee für einen rechteckigen Tisch (nicht-kreisförmig)** Zunächst wird ein numerisches (Integer) Array erzeugt, wobei jeder Arrays Slot die Größe einer am Tisch sitzenden Gruppe enthält. Die Gruppengrößen sollen im Array so angeordnet sein, wie die Gruppen auch am Tisch angeordnet sind. Leere Plätze werden ignoriert.

**Beispiel 14** Ein Tisch der Größe 10 enthalte die folgenden Gruppen.

1	1	1	2	2	0	3	3	3	3
---	---	---	---	---	---	---	---	---	---

(30)

Dann wird das folgende Integer-Array erzeugt.

3	2	4
---	---	---

(31)

Der Spezialfall, dass sich eine Gruppe über die Arraygrenzen erstreckt, kann nicht vorkommen, da der Tisch per Definition nicht kreisförmig ist.

Das Problem kann nun mit dynamischer Programmierung in Polynomialzeit gelöst werden.

**Definition 3** Für eine Instanz des Problems mit  $1 \leq n \leq n_{max}$  Gruppen und  $1 \leq t \leq t_{max}$  Tischteilen kann eine rekursive Formel aufgestellt werden. Dabei sind  $n_{max}$  die Anzahl der Gruppen und  $t_{max}$  die Anzahl der zu erstellenden Tischteile<sup>29</sup>, als noch keine Teilung durchgeführt wurde, also ganz zu Anfang. Als Kosten  $K[n][t]$  sei die die Anzahl der Personen im größten Segment<sup>30</sup> definiert. Die Kosten gilt es zu minimieren, dann sind alle Segmente möglichst gleich groß.

$$K[n][t] = \min_{i=1}^n \max(K[i][t-1], \sum_{j=i+1}^n g_j) \quad (32)$$

$g_j$  ist dabei die Größe der j-ten Gruppe, also der j-te Eintrag im Integer-Array. Anschaulich kann man sich die einzelnen Schritte des Algorithmus wie folgt vorstellen. Es sollen die Kosten der optimalen Lösung für das Problem  $n = n_{max}$  und  $t = t_{max}$  ermittelt werden. Für die letzte Trennung gibt  $n$  Möglichkeiten, den Trennstreich zu setzen. Alle Gruppen auf der rechten Seite des Trennstreiches werden einem Ober zugewiesen. Die Gruppen auf der linken Seite müssen noch aufgeteilt werden. Dabei handelt es sich hierbei um die gleiche Problemstellung, nur dass die Anzahl der Gruppen sich reduziert hat und dass nun ein Trennstreich weniger platziert werden muss. Nun müsste man

<sup>29</sup>Bei  $x - 1$  Teilungen entstehen  $x$  Tischteile.

<sup>30</sup>D.h. die maximale Anzahl an Personen zwischen zwei Teilungen.

einen rekursiven Aufruf für das verkleinerte Problem starten. Alle  $n$  Möglichkeiten, den Trennstrich zu setzen, müssen untersucht werden. Wenn man für jeden solchen Fall einen rekursiven Aufruf starten würde, wäre die Laufzeitkomplexität nicht mehr polynom zur Problemgröße. Deshalb wird die Matrix  $K$  zuerst mit kleinen Fällen aufgefüllt. Alle größeren Fälle können dann auf die Ergebnisse der kleineren Fälle zugreifen und es ist lediglich polynomische Zeit nötig, um die optimale Lösung zu berechnen (weil keine gleichen rekursiven Aufrufe mehrfach erfolgen).

Intuitiv bedeutet die max-Funktion in der Formel folgendes: Entweder ist das größte Segment (mit den meisten Personen) auf der linken Seite der aktuellen Trennung oder das größte Segment ist aktuelle entstehende Segment auf der rechten Seite der Trennung.

Zunächst wird die Matrix mit den Basisfällen initialisiert.

**Definition 4** Für die Matrix  $K$  gelten die folgenden Basisfälle.

Wenn der Tisch nur eine einzige Gruppe enthält, sind die Kosten immer die Größe dieser einzigen Gruppe. Diese darf nämlich nicht zerteilt werden.

$$K[1][1] = K[1][2] = \dots = K[1][t_{max}] = g_1 \quad (33)$$

Wenn der Tisch nur aus einem einzigen Teil bestehen soll, also nicht geteilt werden soll, sind die Kosten dieses einen Tischteiles die Summe aller Gruppengrößen.

$$K[n][1] = \sum_{i=1}^n g_i \quad (34)$$

Dann wird die Matrix so aufgebaut, dass zunächst der Wert  $n$  das Intervall  $[1; n_{max}]$  durchläuft, bis  $t$  um eins erhöht wird. Das liegt daran, dass immer auf die Werte mit dem Wert  $t - 1$  zugegriffen wird.

Um die Kosten des eigentlichen Problems zu ermitteln wird dann der Wert  $K[n_{max}][t_{max}]$  ausgelesen. Die einzelnen Schritte, die zu dieser Lösung führen, können ermittelt werden, indem eine zweite Matrix angelegt wird, die die Entscheidung speichert, die zu den in der Matrix  $K$  angegebenen Kosten führt.

Dieser Algorithmus weist eine Platzkomplexität von  $\mathcal{O}(n_{max} \cdot t_{max})$  auf, weil zwei Matrizen dieser Größe erstellt werden müssen. Die Laufzeitkomplexität beträgt dagegen  $\mathcal{O}(n_{max}^3 \cdot t_{max})$ , weil für jeden Matrixeintrag über alle Tischgrößen ( $n \leq n_{max}$ ) iteriert werden muss, um das Minimum zu finden, und weil für jede Tischgröße über alle Gruppen im letzten Segment iteriert werden muss, um dessen Kosten zu ermitteln. Diese letztere Iteration könnte man sich jedoch sparen, wenn man alle diese Werte in einer Tabelle speichert<sup>31</sup>, was ich jedoch nicht gemacht habe.

**Lösungsidee für einen kreisförmigen Tisch** Für die Lösung dieses Problems kann der zuvor vorgestellte Algorithmus verwendet werden. Besteht der Tisch aus  $n_{max}$  Gruppen und sollen  $t_{max}$  Unterteilungen vorgenommen werden, kann das Problem gelöst werden,

<sup>31</sup>Das Erstellen der Tabelle hängt quadratisch von der Tischgröße ab (es müssen alle Werte von  $j$  und  $n$  - siehe Formel - betrachtet werden).

indem der vorherige Algorithmus mit den  $n_{max}$  Gruppen, aber mit  $t_{max} - 1$  Unterteilungen aufgerufen wird. Die letzte Unterteilung stellen die Arraygrenzen dar. Um nun alle Möglichkeiten zu generieren, diese letzte Unterteilung zu verschieben, wird das gesamte Gruppenarray um 1 verschoben, d.h. die Reihenfolge der Elemente untereinander bleibt gleich, aber das Element auf Arrayslot 1 ist nun auf Arrayslot 2, usw. Für diese Verschiebung wird nun der Algorithmus aufgerufen. Dann wird das Array wieder um 1 verschoben und es wird wieder der Algorithmus aufgerufen. Dieser Vorgang wird insgesamt  $n_{max}$  Mal wiederholt. Die beste Lösung wird gespeichert und ausgegeben.

Die Laufzeitkomplexität des Algorithmus erhöht sich durch diesen Schritt auf  $\mathcal{O}(n_{max}^4 \cdot t_{max})$ , weil der Algorithmus nun  $n_{max}$  Mal aufgerufen wird.

### 1.3.5 Normale Personen

Neben den Schülern besuchen auch *normale* Personen das Restaurant. Es gibt eine bestimmte Anzahl an freien normalen Personen, die das Restaurant besuchen können. Das Programm entscheidet zufällig, welche Gruppen erstellt und entfernt werden sollen. Es wird auch zufällig entschieden, wie groß die Gruppen sein sollen. Diese Erweiterung ist sinnvoll, weil es völlig selbstverständlich ist, dass das Restaurant auch von normalen Personen besucht wird. Man kann jetzt untersuchen, wie sich (scheinbar zufällige) Entscheidungen von anderen Personen im Restaurant auf die Funktionsweise des Algorithmus auswirken, also wie der Algorithmus im Normalfall funktioniert.

Vom Ober werden Gruppen, bestehend aus normalen Personen, wie Schülergruppen behandelt, es wird der gleiche Algorithmus angewendet, um die Gruppen möglichst platzsparend zu platzieren. Der Koordinator behandelt normale Gruppen wie feste Gruppen. Sie werden also auch beim Algorithmus, der entscheidet, welche Gruppen entfernt werden sollen, miteinbezogen. Möglicherweise werden deshalb weniger Schüler benötigt. Da aber die normalen Gruppen zufällig wieder aufgelöst werden, kann es sein, dass dadurch große Lücken entstehen, die dazu führen, dass der Algorithmus des Koordinators nicht mehr richtig funktioniert. Neu erstelle Gruppen passen dann möglicherweise in die Lücken, die entstanden sind, als eine normale Gruppe aufgelöst wurde. Im Kapitel *Programm-Ablaufprotokoll* gibt es Beispiele zu sehen.

## 2 Programm-Dokumentation

Das Programm wurde in C# (Microsoft .NET) geschrieben und benötigt das Microsoft .NET Framework 3.5 SP1. Das Programm schaut an manchen Stellen noch nicht ganz *fertig* aus. Das liegt daran, dass ich eigentlich noch viel mehr Funktionen einbauen wollte, an manchen Stellen habe ich das Programm auch so geschrieben, dass Erweiterungen möglichst einfach einzubauen sind. Zum Schluss war dann doch wieder nicht genug Zeit.

### 2.1 Aufbau des Programms

Bei der Entwicklung des Programms wurde sorgfältig darauf geachtet, Programmoberfläche und Algorithmen zu trennen. Das kam mir dann besonders beim Debuggen zu

Gute. In der Regel beginnen Klassennamen, Variablenbezeichnungen oder ähnliches mit dem Prefix *GUI* oder die entsprechenden Klassen befinden sich im Namespace *GUI*, wenn sie Teil der Programmoberfläche sind.

Die Klasse *Restaurant* bündelt den Zugriff auf alle Objekte im Restaurant. So enthält sie zum Beispiel Zeiger auf alle Tische im Restaurant<sup>32</sup>, sowie Zeiger auf den Gruppenkoordinator und auf den Chefober. Ebenso enthält die Klasse ein *Timer*-Objekt, mit dem sich die Geschwindigkeit der Simulation anpassen lässt. Das *Tick*-Ereignis des Timers tritt nach einem bestimmten Intervall-Wert ein und wird an alle Ober, Chefober und Gruppenkoordinator weitergegeben. Außerdem vergeht mit jedem Tick-Ereignis eine Zeiteinheit<sup>33</sup>.

Jeder Tisch enthält ein Platzarray, welches für jeden Sitzplatz die zugewiesene Gruppe speichert. Dafür gibt es für jede Gruppe eine eigene Klasse. Außerdem existiert für jeden Tisch ein Zeiger auf den zuständigen Ober.

Alle Zeiger von *Restaurant* und *Tisch* auf *Ober*, *Chefober*, *Gruppe* und *Gruppenkoordinator* wurden über Interfaces realisiert. Dadurch können neue Algorithmen schnell ausprobiert werden. Die entsprechenden Klassen müssen nur die Interfaces implementieren. Außerdem können so verschiedene *Chefober* und *Koordinatoren* gegeneinander antreten.

### 2.1.1 Der einfache Chefober

Aufgabe des *Chefobers* ist es, einzelne Aufgaben an verschiedene *Ober* (und damit *Tische*) zu delegieren. Wenn beispielsweise eine neue Gruppe kommt, entscheidet der *Chefober*, an welchen Tisch die Gruppe kommt. Der *einfache Chefober* hat also nicht viel zu tun, da er nur einen einzigen Tisch verarbeiten kann. Er übergibt die ganze Arbeit an den einzigen *Ober*. Wenn eine neue Gruppe im Restaurant ankommt, wird beim *Chefober* das Ereignis *GruppeKommt* ausgelöst. Daraufhin wird die Gruppe am Ende einer Dequeue (Double-Ended-Queue) eingefügt. Jedes Mal, wenn ein *Tick*-Ereignis ausgelöst wird, übergibt der *Chefober* alle Gruppen in der Dequeue an den *Ober*, und zwar in der Reihenfolge, in der sich die Gruppen in der Dequeue befinden. Gruppen, die zuerst gekommen sind, kommen also früher dran. Der einfache *Ober* versucht dann, die Gruppe zu platzieren. Falls das nicht erfolgreich war, wird die Gruppe wieder am Ende der Dequeue eingefügt.

### 2.1.2 Der einfache Ober

Für jeden Tisch existiert ein *Ober*. Dessen Aufgabe ist es, Gruppen an seinem Tisch zu platzieren. Gelingt ihm dies nicht, teilt er das dem *Chefober* mit. Dieser versucht dann beim nächsten *Tick*-Ereignis nochmals, die Gruppe an einen Tisch mit ausreichend Platz zu delegieren. Im Kapitel *Lösungsidee* wurde bereits erklärt, wie der *einfache Ober* die Gruppen platziert, nämlich so, dass möglichst wenig Platz verschwendet wird.

---

<sup>32</sup>Das Programm wurde so erweitert, dass mehrere Tische eingerichtet werden können.

<sup>33</sup>Das ist besonders für Debugzwecke interessant.

Dazu wird der Algorithmus *SubarrayMitMindestlaenge* benötigt. Dieser Algorithmus versucht, im Platzarray eines Tisches ein möglichst kleines freies Intervall<sup>34</sup> zu finden, das jedoch eine gewisse Mindestgröße aufweist. Diese Mindestgröße entspricht der Größe der zu platzierenden Gruppe. Der Algorithmus gibt immer den Anfang (Index) der Lücke zurück und die Gruppe wird immer am Anfang der Lücke platziert. Falls die Lücke also größer als die Gruppengröße ist, wird die Gruppe *nicht* in der Mitte der Lücke platziert, sondern ganz am Anfang der Lücke, um *eine* möglichst große Restlücke zu erhalten.

Der Algorithmus untersucht zuerst, ob die Mindestgröße (der Lücke) größer als das Intervall ist, in einem solchen Fall kann es nämlich keine derartige Lücke geben und es wird *-1* zurückgegeben. Die Variable *aktuellePos* speichert die Stelle, an der die aktuelle Lücke beginnt. Ganz zu Anfang steht diese Variable also auf *-1*, weil noch kein freier Platz gefunden wurde. Selbiges gilt für *bestePos*, nur dass diese Variable für die beste, bereits gefundene Lücke gilt, also der Lücke, die möglichst klein ist und die geforderte Mindestgröße aufweist. Der Algorithmus iteriert nun zweimal über das Platzarray<sup>35</sup>. Die zweite Iteration ist deshalb notwendig, weil es sich um einen runden Tisch handelt. Wenn nun auf einen Platz zugegriffen wird, geschieht dies immer Modulo Arraygröße. Somit können auch Lücken erfasst werden, die sich über die Arraygrenze erstrecken (die also z.B. mitten im Array beginnen und so groß sind, dass wieder von vorne weiteriteriert werden muss). Eine Lücke ist beendet, wenn nach einer Folge von freien Plätzen wieder ein besetzter Platz gefunden wird. Dann wird auch die Größe der Lücke ermittelt und mit der aktuell besten Lücke verglichen.

Die Position, an der der Algorithmus mit dem Durchsuchen des Platzarrays beginnt, ist *startPos*. Dabei handelt es sich um den ersten unbesetzten Platz (Grund siehe Lösungs-idee). Gibt es keinen solchen Platz, ist der Tisch komplett leer und die Gruppe wird ab dem ersten Sitzplatz platziert.

### 2.1.3 Die einfache Gruppe

Wie bereits erwähnt, wird jeder Gruppe eine eigene Klasseninstanz zugewiesen. Für jede Gruppe werden der Tisch, an dem sich die Gruppe befindet, und das Platzintervall gespeichert. Dazu wird die Struktur *GruppePositionAmTisch* verwendet. Außerdem wird eine Gruppe (wie jedes andere Objekt) darüber informiert, wenn eine Zeiteinheit vergangen ist. Wenn eine Gruppe nach (standardmäßig) zwei vergangenen Zeiteinheiten noch immer nicht platziert wurde, so verlässt sie das Restaurant und der Ober wird sich ärgern. Diese Stelle lässt Freiraum für eigene Erweiterungen, so könnte es zum Beispiel besonders reiche und wichtige Gruppen geben, die nicht so lange warten wollen<sup>36</sup>, aber später mehr dazu.

---

<sup>34</sup>Ein Platz ist frei, wenn die Funktion *array* für einen Platzindex den Wert *wahr* zurückgibt. Bei *array* handelt es sich also um einen Funktionszeiger (Delegate), der auf die boolsche Funktion *Tisch.IstPlatzFrei(platznummer)* zeigt.

<sup>35</sup>Daraus ergibt sich eine Laufzeitkomplexität von  $\mathcal{O}(n)$ , wenn *n* die Größe des Platzarrays ist.

<sup>36</sup>Damit würde man aber die Forderung aus der Aufgabenstellung, dass Gruppen immer sofort platziert werden sollen, aufweichen.

### 2.1.4 Der einfache Gruppenkoordinator

Der Koordinator soll entscheiden, wie sich die Gruppen aus den freien Personen zusammensetzen und wann Gruppen das Restaurant verlassen sollen. Wie bereits gesagt, warten Gruppen zwei Zeiteinheiten, bis sie das Restaurant verlassen, wenn sie keinen Platz bekommen. Um hier nicht in Konflikt mit der Aufgabenstellung zu geraten, wartet der Koordinator so lange, bis die letzte erstellte Gruppe platziert wurde, bevor eine neue Aktion ausgeführt wird<sup>37</sup>. Der Gruppenkoordinator speichert die Liste mit freien Personen und ebenso eine Liste mit allen aktuell existierenden Gruppen. Wenn das Tick-Ereignis beim Koordinator eintritt, wird zuerst überprüft, ob die letzte erstellte Gruppe schon platziert wurde. Wenn das nicht der Fall ist, wird nichts gemacht, bis das nächste Tick-Ereignis eintritt. Ansonsten verfährt der Koordinator wie folgt.

Eine numerische Variable *\_internerZustand* mit dem Standardwert 0 gibt an, in welchem *Verarbeitungsschritt* der Koordinator sich befindet. Wenn der interne Zustand auf Null steht, beginnt der Koordinator damit, so viele Gruppe der Größe 1 zu erstellen, wie Personen vorhanden sind. Dann wird der Zustand auf 1 gesetzt und auf das nächste Tick-Ereignis<sup>38</sup> gewartet. Wenn der interne Zustand 1 oder größer ist, beginnt der Koordinator damit, Gruppen zu entfernen und größere Gruppen zu erzeugen. Im Kapitel *Lösungsidee* war bereits die Rede von einer Folge  $f_n$ , die angibt, wie groß die Gruppen im aktuellen Verarbeitungsschritt sein sollen. Der Koordinator versucht immer, möglichst viele Personen zu entfernen, wobei keine Lücken größer als  $f_{\_internerZustand} - 1$  (maximale Lückengröße) entstehen dürfen, und Gruppen der Größe  $f_{\_internerZustand}$  zu erzeugen. Das erste Element der Folge ist nach Definition  $f_1$ . Um zu entscheiden, welche Gruppen gelöscht werden sollen, wird der Algorithmus *EinfachesRestaurant\_EntferneGruppen* angewendet, der später genau erklärt werden soll. Nachdem die Gruppen entfernt wurden, gibt es zwei Möglichkeiten fortzufahren. Entweder sind jetzt schon genügend kleine Lücken vorhanden, um den Ober zu ärgern (Fall 1) oder das Platzarray muss noch weiter zerstückelt werden (Fall 2). Fall 1 trifft zu, wenn die größte Lücke am Tisch kleiner als die Anzahl der freien Personen ist. Um die Größe der größten Lücke zu ermitteln, wird der Algorithmus *LaengstesSubarray* verwendet, der ebenfalls später erklärt werden soll. Bei Fall 1 genügt es nun, einfach eine große Gruppe mit allen freien Personen (oder *größte Lückengröße + 1*) zu erzeugen. Wenn Fall 1 nicht zutrifft, so trifft Fall 2 zu. Dann werden aus den freien Personen neue Gruppen gebildet, die die Größe  $f_{\_internerZustand}$  aufweisen. An dieser Stelle kann der Algorithmus optional noch leicht verändert werden<sup>39</sup>: Es besteht die Möglichkeit, die letzte Gruppe etwas größer zu machen, wenn die restlichen freien Personen nicht mehr ausreichen, um eine neue Gruppe der Größe  $f_{\_internerZustand}$  zu erzeugen. Im Kapitel *Ablaufprotokoll* wird sich zeigen, ob und warum das sinnvoll ist oder nicht.

---

<sup>37</sup>So kann es zum Beispiel nicht dazu kommen, dass eine Gruppe erst dann platziert wird (werden kann), nachdem eine weitere Gruppe entfernt wurde, obwohl sich der Ober schon vorher hätte ärgern müssen.

<sup>38</sup>Eigentlich darauf, dass alle Gruppe platziert wurden

<sup>39</sup>In der GUI kann diese Option aktiviert oder deaktiviert werden.

### 2.1.5 Der Algorithmus EinfachesRestaurant\_EntferneGruppen

Aufgabe des Algorithmus ist es, zu entscheiden, welche Gruppen entfernt werden sollen, ohne dass zu große Lücken entstehen. Als Eingabe erhält der Algorithmus eine Tisch-Klasse. Ich vermute, dass es (u.a. im Laufzeithalten) besser ist, mit einfachen Strukturen und Wertetypen zu arbeiten, anstatt mit komplexen Klassen, bei denen man auch immer das Problem Referenz-/Wertetyp beachten muss. Deshalb wird die Tisch-Klasse in eine Tisch-Kosten-Struktur umgewandelt, bevor der Algorithmus beginnt. Während eine Tisch-Klasse für jeden Sitzplatz einen Zeiger auf eine *IGruppe-Klasseninstanz* speichert, verwendet die Struktur Zahlenwerte, wobei jeder Gruppe ein eindeutiger Index zugewiesen wird. Dieser Index wird bereits beim Instanziieren der Tisch-Klasse erzeugt. Es wird so sichergestellt, dass zwei Klassen nie den gleichen Index bekommen. Leere Plätze haben den Index Null und feste bzw. normale Gruppen haben immer negative Indizes<sup>40</sup>. Jede Tisch-Kosten-Struktur speichert außerdem noch die maximale Lückengröße<sup>41</sup> sowie die Kosten des aktuellen Tisches. Die Kosten ergeben sich aus der Summe der Kosten der einzelnen Personen, wobei jede Person standardmäßig eine Kosteneinheit verursacht<sup>42</sup>. Wenn im Laufe des Algorithmus versucht wird, eine gute Lösung zu finden, wird die Lösung ausgewählt, die die geringsten Kosten verursacht<sup>43</sup>.

Bevor der Algorithmus starten kann, muss noch ein boolsches Array *original\_kosten* der Größe des Tisches (Anzahl der Sitzplätze) erzeugt werden. Dieses Array steht genau dann auf *wahr*, wenn eine Sitzplatzlücke am Tisch bereits jetzt (vor Anwendung des Algorithmus) zu groß ist, also die *maximale Lückengröße* überschreitet. Dazu wird der Algorithmus *BerechneOriginalKosten* verwendet, der später genauer erklärt wird.

Leider gibt es beim eigentlichen Algorithmus zwei verschiedene Fälle, weshalb zwei Funktionen erstellt wurden. Diese Funktionen haben beide den gleichen Namen (*EinfachesRestaurant\_EntferneGruppen*) und unterscheiden sich lediglich in ihren benötigten Parametern (Überladungen). Der Algorithmus geht nach dem *Divide-and-Conquer-Verfahren* vor, das Sitzplatzarray wird also immer weiter unterteilt. Vor der ersten Unterteilung handelt es sich noch um den gesamten Tisch (Fall A), deshalb muss hier beachtet werden, dass es sich um einen runden Tisch handelt. Das erfordert ein paar kleinere Änderungen am Algorithmus. Nach der ersten Unterteilung muss das nicht mehr beachtet werden, da nur ein kleiner Ausschnitt des Tisches betrachtet wird (Fall B), der keinen geschlossenen Kreis darstellt. Es muss also beispielsweise nicht der Spezialfall betrachtet werden, dass sich eine Lücke an den Arraygrenzen befindet und diese überschreitet (dann müsste wieder am Anfang weiteriteriert werden).

Im ersten Schritt des Algorithmus trifft noch Fall A zu. Es gilt das Problem zu behandeln, dass nach Abschluss des Algorithmus möglicherweise eine zu große Lücke an den Arraygrenzen existiert. Deshalb müssen hier die zwei Entscheidungen *Gruppe ganz*

---

<sup>40</sup>Beim Instanziieren wird ein positiver Index vergeben, der vom Algorithmus dann mit  $-1$  multipliziert wird.

<sup>41</sup>Dieser Begriff wird synonym zu *maximale Intervallgröße* verwendet.

<sup>42</sup>Das Programm kann an dieser Stelle erweitert werden, indem man verschiedenen Gruppen/Personen unterschiedliche Kosten zuweist.

<sup>43</sup>Bei dieser Lösung wurden die meisten Personen entfernt.

*links entfernen oder nicht* und *Gruppe ganz rechts entfernen oder nicht* (also vier Fälle), getroffen werden. Es werden alle vier Fälle ausprobiert und das beste Lösung (mit den geringsten Kosten), die auch noch gültig ist, wird gewählt. Es wird also für jeden der vier Fälle der eigentliche Algorithmus gestartet. Wenn eine Gruppe nicht entfernt wird, wird sie als *fest* markiert. Ihr eindeutiger Index wird invertiert (negativ machen). Somit existiert mit Sicherheit ein Fall, der gültig ist, nämlich der, bei dem keine Gruppe entfernt wird. Wenn keine Änderungen vorgenommen werden bleibt der Fall logischerweise gültig. Bei der Prüfung auf Gültigkeit werden zu große Lücken, die schon vorher zu groß waren, nicht beachtet, deshalb wird das boolsche Array *original\_kosten* benötigt. Für jeden der vier Fälle wird nun der Algorithmus für Fall B angewendet. Zum Schluss werden für jeden der vier Fälle die Gültigkeit und Kosten überprüft.

Die Teilung des Sitzplatzarrays (Fall B<sup>44</sup>) sollte immer möglichst in der Mitte stattfinden. Nur dann ergibt sich die erwartete Laufzeitkomplexität aus dem Kapitel *Lösungs-Idee*. Um eine geeignete Stelle zur Trennung zu finden, wird von der Mitte des Arrays aus zum linken Rand iteriert. Es wird dann so getrennt, dass keine Gruppe auseinandergerissen wird. Falls keine solche Stelle existiert, wird zum rechten Rand iteriert. Wenn auch jetzt noch keine passende Stelle gefunden werden kann, besteht das Array aus nur einer einzigen Gruppe, die das ganze Array ausfüllt. Dieser Spezialfall wird schon ganz zu Anfang überprüft. Wenn das ganze Array aus nur einer Gruppe besteht, muss diese notwendigerweise eine feste Gruppe sein (weil die Gruppen an den Rändern immer entweder entfernt oder als fest markiert werden) und es kann abgebrochen werden, weil es nichts zu tun gibt. Selbiges trifft natürlich auch zu, wenn das Platzarray keine einzige entfernbare (nicht feste) Gruppe enthält. Nach der Trennung gibt es wieder die vier Entscheidungsmöglichkeiten (siehe *Lösungs-Idee*). Bevor nun vier rekursive Aufrufe gestartet werden, wird das Array zuvor auf Gültigkeit überprüft, um sich ggf. Arbeit sparen zu können, nämlich dann, wenn nach dem Entfernen von Gruppen an den Arraygrenzen der Fall schon ungültig geworden ist. Nach dem rekursiven Aufruf wird das Array nochmals geprüft und wieder zusammengesetzt. Die beste Lösung (Entscheidungsmöglichkeit) wird zurückgegeben.

### 2.1.6 Prüfen eines Platzarrays auf Gültigkeit

Der Algorithmus erwartet als Eingabe eine Tisch-Kosten-Struktur und ein boolsches Array *original\_kosten*. Es wird über das gesamte Sitzplatzarray iteriert und immer, wenn ein leerer Platz gefunden wurde, wird der Zähler, der die aktuelle Lückengröße speichert, um eins erhöht. Wenn ein leerer Platz gefunden wird und zudem das Array *original\_kosten* an dieser Stelle noch auf *wahr* steht, wird der Zähler nicht erhöht, da es sich um eine Lücke handelt, die schon vor Anwendung des Algorithmus zu groß war. Wenn der Algorithmus wieder auf einen besetzten Platz trifft (oder der Algorithmus am Ende des Arrays angekommen ist), wird der Zähler ausgewertet. Falls auf eine begonnene Lücke (die vorher noch nicht da war) ein freier Platz folgt bei dem das boolsche Array auf *wahr* steht, folgt, wurde eine zu große Lücke weiter vergrößert und der Fall ist ungültig.

---

<sup>44</sup>Bei Fall A findet gar keine Teilung statt, es werden nur die äußeren Gruppen betrachtet und das Ergebnis ganz zum Schluss überprüft.

Falls das Array noch nicht als üngültig erkannt worden ist, wird in die andere Richtung iteriert (von rechts nach links statt von links nach rechts). Somit kann man ausschließen, dass eine bestehende, zu große Lücke in die andere Richtung weiter vergrößert worden ist. Man könnte den Algorithmus auch so anpassen, dass nur eine Iteration benötigt wird, was aber keinen signifikanten Einfluss auf das Laufzeitverhalten hat.

Falls das Array *mit Umlauf* überprüft werden soll (Fall A trifft zu), muss der Algorithmus so verändert werden, dass die Iterationsgrenze bei der doppelten Arraygröße liegt (es wird also zweimal über das Array iteriert). Alle Zugriffe auf das boolsche Array oder das Sitzplatzarray geschehen dann Modulo Arraygröße.

### 2.1.7 Berechnung des boolschen Arrays `original_kosten`

Die Berechnung des boolschen Arrays erfolgt *mit Umlauf*, es wird also beachtet, dass der Tisch rund ist. Es ist kein Algorithmus notwendig, der dies zusätzlich *ohne Umlauf* macht, weil bei einem rekursiven Aufruf des Algorithmus *EntferneGruppen* das boolsche Array ebenfalls geteilt wird und unverändert an den rekursiven Aufruf weitergegeben wird. Es sollen schließlich nur die Lücken ignoriert werden, die schon vor Anwendung des Algorithmus zur groß waren<sup>45</sup>.

Um die zu großen Lücken zu finden, wird wieder zweimal über das Sitzplatzarray iteriert. Der Zugriff auf die Sitzplätze erfolgt wieder Modulo Arraygröße. Wenn der erste leere Sitzplatz gefunden wird (oder der erste leere Sitzplatz nach einem Intervall besetzter Plätze), beginnt ein *Lauf*. Wenn nun wieder auf einen besetzten Platz getroffen wird, wird die Länge des aktuellen Laufes untersucht. Ist diese zu groß, wird das boolsche Array zwischen Laufbeginn und Laufende auf *wahr* gesetzt.

## 2.2 Eigene Erweiterungen

### 2.2.1 Der Algorithmus des perfekten Obers

Der Algorithmus des perfekten Obers befindet sich in der Klasse *MinimaxOber*. Der Name kommt daher, dass ich ursprünglich einen Minimax-Algorithmus erstellen wollte. Im Nachhinein wollte ich nicht alle Namen austauschen.

Für diesen Algorithmus wurde eine Bibliothek von Adrian Akison von CodeProject.com verwendet, die kombinatorische Probleme lösen kann (Generieren aller Permutationen etc.). Es wurden eine neue Klasse und eine neue Struktur erstellt, die vom Algorithmus verwendet werden. Die Klasse *SpielBaumKnoten* stellt einen Spielzustand dar (siehe Lösungsidee). Es handelt sich dabei nicht wirklich um einen Baum oder einen Knoten, denn es werden keine Zeiger auf das nächste Element oder das vorherige Element gespeichert, ebenso werden die Klasseninstanzen gelöscht, sobald sie nicht mehr gebraucht werden. Man kann sich die einzelnen Spielzustände aber wie einen Baum vorstellen<sup>46</sup>, in Algorithmen-Büchern findet man das oft so vor.

---

<sup>45</sup>In einer sehr frühen Version des Programms hatte ich einen Denkfehler begangen und das boolsche Array bei jedem rekursiven Aufruf neu berechnet.

<sup>46</sup>Dann sieht man wie schnell die Anzahl der Spielzustände (meist exponentiell) zunimmt.

Die Funktion *ErmittleGewinner* untersucht, welcher Spieler bei einem vorliegenden Spielzustand gewinnt: entweder der einfache Koordiantor oder der perfekte Ober. Dabei wechseln sich rekursive Aufrufe für den Ober und den Koordinatator ab. Wenn ein rekursiver Aufruf für den Koordinatator auftritt, wird ein entsprechender If-Zweig aufgerufen, der Code-Stücke des einfachen Koordinators enthält und sich genauso verhält, wie der einfache Koordinatator (siehe Kapitel *Einfacher Koordinatator*). Wenn hingegen der Ober am Zug ist, werden alle Spielzüge generiert. Das Prinzip wurde schon bei der Lösungs-idee vorgestellt. Zum Generieren aller Variationen/Permutationen wurde die erwähnte externe Klasse verwendet.

Es wurde eine Struktur *ZustandGewinner* erstellt, die die Funktion *ErmittleGewinner* zurückgibt. Diese Struktur enthält den Gewinner in der aktuellen Situation und eine Liste mit allen erfolgten Spielzügen. Liste und Gewinner werden dabei *rückwärts* aufgebaut. Das Ergebnis wird erst von der Funktion zurückgegeben, wenn alle rekursiven Aufrufe erfolgt sind und ausgewertet wurden. Es werden bei einem Aufruf der Funktion die Elemente der Liste eines rekursiven Aufrufs übernommen. Dabei handelt es sich um den rekursiven Aufruf, der zum Sieg führt, wenn es einen solchen gibt. Für den Fall, dass es keinen siegbringenden Spielzug für den Ober gibt, ist die Liste der Spielzüge also leer.

### 2.2.2 Reservierungen im Turmrestaurant

Der Algorithmus für diese Erweiterung befindet sich in der Klasse *OberGaesteAuswahl*. Bevor der Algorithmus gestartet werden kann, muss eine neue Instanz dieser Klasse erstellt werden. Dann kann die Funktion *GastHinzufuegen* benutzt werden, um einzelne Reservierungswünsche hinzuzufügen. Der Algorithmus wird über die Prozedur *ErmittleBesteAuswahl* gestartet und gibt eine Liste der angenommenen Reservierungen zurück.

Es existieren zwei Matrizen, die den maximalen Profit und die dazu führende Entscheidung für jeden Fall speichern. Nutzungsgrenzen gibt es prinzipiell keine, wenn man von Integer-Überläufen bei zu vielen Eingabedaten absieht. Auch die Laufzeit sollte keine Nutzungsgrenze darstellen (polynomielle Laufzeit). Die Erweiterung kann im Hauptfenster des Programms im Menü Restaurant/Erweiterungen gestartet werden.

Das Ermitteln der einzelnen Schritte mit der zweiten Matrix erfolgt analog zur Erweiterung *Zuständigkeit einzelner Kellner* (siehe weiter unten), dieses Problem wurde nämlich auch mit DP gelöst.

### 2.2.3 Kontostand für einzelne Schüler

Es soll nun kurz darauf eingegangen werden, welche Anpassungen notwendig waren, um sich möglichst viel Code zu sparen, indem objektorientierte Programmierung verwendet wurde. Die neue Klasse *KoordinatorGeld* leitet von der Klasse *EinfacherKoordinatator* ab und überschreibt einige Funktionen, sodass nun für jede freie Person der Kontostand überprüft wird. Die Eigenschaft *AnzahlDerPersonen* wurde überschrieben, um den Container *FreiePersonen* mit einem anderen Typ zu initialisieren. *FreiePersonen* muss lediglich das (generische) Interface *IObjectContainer* implementieren. Jede solche Klasse muss die Funktionen *HoleErstesElement*, *FuegeElementEin* und *Count* implementieren.

Beim einfachen Koordinator wurde dazu die Klasse *ObjectList* verwendet, die intern mit einer Liste arbeitet. Jedes neue Element wird am Ende der Liste eingefügt und wenn ein Element entnommen werden soll, wird immer das erste Element entfernt. Bei der Klasse *KoordinatorGeld* wird hingegen der Container *ObjectHeap* verwendet, der intern mit dem selbst geschriebenen Heap arbeitet. Wenn ein Element entnommen wird, wird somit immer das maximale Element entnommen.

Immer wenn eine Funktion aus *KoordinatorGeld* oder der Basisklasse des einfachen Koordinator eine freie Person anfordert, wird somit die Person mit dem maximalen Kontostand entnommen. Nun muss noch sichergestellt werden, dass Personen ohne Geld automatisch aussortiert werden. Wenn eine Person zum Container der freien Personen hinzugefügt werden soll, wird die Funktion *FreiePersonHinzufuegen* genutzt. Auch diese wurde überschrieben und überprüft, ob die freie Person überhaupt Geld hat<sup>47</sup>. Wenn das nicht der Fall ist, wird sie erst gar nicht zum Container hinzugefügt. Analog wird die Funktion *FreiePersonHolen* überschrieben, damit bei jeder freien Person, die entnommen wird, der Kontostand um 1 verringert wird. Ansonsten brauchen keine weitere Funktionen überschrieben werden, die geerbten Basisfunktion verwenden automatisch die neuen, überschriebenen Funktionen.

## 2.2.4 Zuständigkeit einzelner Kellner

Der eigentliche Algorithmus befindet sich in der Klasse *KellnerVerteilen*. Die Erweiterung kann über das Menü aufgerufen werden. Bevor der Algorithmus gestartet werden kann, muss die Liste *\_tischMitGruppen*, die alle Gruppen (Reihenfolge ist wichtig!) enthält, gesetzt werden. Ebenfalls müssen *\_tMax* und *\_nMax*<sup>48</sup> gesetzt werden, wobei diese Variablen den Variablen aus der Lösungsidee entsprechen. Wenn die Funktion *Start* aufgerufen wird, beginnt der Algorithmus. Als Rückgabe wird eine Liste mit den Indizes, an denen geteilt werden soll, übergeben.

Der eigentliche Algorithmus wird einige Male nacheinander ausgeführt. Dazu ist es notwendig, das Gruppenarray (Liste der Gruppen, *\_tischMitGruppen*) um eine Position zu verschieben. Zuerst wird der erste Eintrag in der Liste ausgelesen, dann wird jedes Element bis auf das Letzte durch seinen Nachfolger ersetzt. Das letzte Element wird dann mit dem gespeicherten Element (ehemals auf erster Position) ersetzt. Bevor der eigentliche Algorithmus gestartet werden kann, muss die Funktion *GeneriereMatrixBasisfaelle* ausgeführt werden, die die Matrizen dimensioniert und die Basisfälle wie in der Lösungsidee angegeben setzt. Der eigentliche Algorithmus startet, wenn die Funktion *DerAlgorithmus* aufgerufen wird. Der Algorithmus schreibt die Ergebnisse (Lösung) in lokale Variablen. Diese werden dann überprüft und wenn die Kosten geringer sind, als beim zur Zeit besten Fall, wird der Fall gespeichert, solange bis ein besserer gefunden wird.

<sup>47</sup>Es existiert ein Interface *IPerson*, das alle Personen implementieren müssen. Die Klasse *PersonGeld* speichert zusätzlich noch einen Kontostand. Zu Fehlern kann es beim Casten (Typumwandlung) auf *PersonGeld* nicht kommen, weil der *KoordinatorGeld* nur *PersonGeld*-Klassen zum Container hinzufügt.

<sup>48</sup>Diesen Wert könnte man auch aus der Anzahl der Gruppen in der Liste ablesen.

### 2.2.5 Normale Personen

Diese Erweiterung befindet sich in der Klasse *NormalePersonen*. Die Klasse übernimmt dabei die Aufgabe des Koordinators, es wird entschieden, welche Gruppen gebildet und entfernt werden sollen. Nur geschieht das zufällig. Die *OnTick*-Prozedur wird ausgelöst, wenn eine Zeiteinheit vergangen ist. Wenn die zuletzt erstellte Gruppe noch nicht platziert wurde, wird gleich abgebrochen. Jetzt wird zufällig entschieden, wie viele Gruppen entfernt werden sollen (*anzahlGruppen*). Auch die Indizes der zu entfernenden Gruppen werden zufällig ausgewählt. Danach wird entschieden, wie viele Personen bei der Bildung neuer Gruppen verwendet werden sollen. Es gibt einen Zähler *anzahlDerPersonen* der angibt, wie viele freie normale Personen zur Verfügung stehen. Beim Entfernen und Erstellung von Gruppen wird dieser entsprechend verändert. Die Anzahl der im aktuellen Durchlauf zu verwendenden Personen (bei der Bildung von Gruppen) ist *anzahlPersonen*. Eine While-Schleife läuft jetzt so lange, bis *anzahlPersonen* auf Null steht. In jedem Durchlauf der While-Schleife wird eine neue Gruppe erstellt. Die Gruppengröße wird dabei zufällig gewählt (zwischen 1 und *anzahlPersonen*). Nachdem die Gruppe erzeugt wurde und dem Chefober übergeben wurde, wird der Wert *anzahlPersonen* verringert.

Es wurde außerdem eine neue Klasse *NormaleGruppe* erstellt, die von der Klasse *EinfacheGruppe* ableitet. Es werden dabei nur zwei Eigenschaften überschrieben: Der Wert *IstSchuelerGruppe*, sodass der Algorithmus des einfachen Koordinators weiß, dass es sich um eine normale (fest) Gruppe handelt und die Farbe *GuiGruppenFarbe*, sodass jede normale Gruppe in der GUI in Dunkelgrau erscheint. Somit können normale Gruppen von Schülergruppen entschieden werden.

**Der eigentliche Algorithmus** Die beiden äußeren For-Schleifen iterieren über alle Tischgrößen (Anzahl der Gruppen) und Anzahl der Unterteilungen. Für jeden solchen Fall wird dann die beste Lösung für den Matrixslot  $K[n][t]$  (*\_matrixK[n, t]*) gesucht. Dazu iteriert das Programm wieder über alle Tischgrößen (das entspricht dem min in der Formel) und sucht die beste Lösung. Wenn eine bessere Lösung gefunden wurde, werden beide Matrizen aktualisiert. Die zweite Matrix speichert die Größe des Restproblems, das ist die Stelle, an der getrennt wurde. So können später die Entscheidungen, die zu dieser Lösung geführt haben, wiedergefunden werden. Dazu wird eine While-Schleife verwendet, die zuerst den Matrixeintrag  $K[n_{max}][t_{max}]$  untersucht. Es wird die Stelle, an der getrennt wurde, ausgelesen und dann wird der Matrixeintrag für dieses Problem gesucht (Anzahl der Teilungen muss ebenfalls reduziert werden). Wenn das Restproblem keine Gruppen mehr enthält, terminiert die While-Schleife. Die Liste der Indizes der Trennungen wird zurückgegeben.

## 2.3 Double-Ended-Queue (Dequeue)

Da das .NET Framework keine fertige Implementation für eine Dequeue bereitstellt, habe ich eine eigene generische Klasse *Dequeue* erstellt. Die generische Klasse *DequeueItem* stellt ein Element in der Dequeue dar und enthält Zeiger auf das vorherige und auf

das nächste *DequeueItem*<sup>49</sup>. Außerdem wird natürlich noch das eigentliche Element (der Datensatz) gespeichert. Die Klasse *Dequeue* unterstützt die Operationen *am Anfang einfügen*, *am Ende einfügen*, *am Anfang entnehmen*, *am Ende entnehmen* und *Größe ermitteln*. Es werden in der Klasse *Dequeue* lediglich zwei Zeiger gespeichert, nämlich auf das erste und das letzte Element. Wenn die Datenstruktur noch leer ist, handelt es sich bei diesen zwei Zeigern um Nullzeiger<sup>50</sup>.

## 2.4 Max-Heap

Der Heap ist eine Datenstruktur, die Elemente aufnehmen kann und sie dabei sortiert. Das Entfernen des größten Elements ist dann sehr schnell (effizient) möglich. Prinzipiell wäre es auch möglich, den Wert eines Elements zu ändern und dann neu einzusortieren, was in meiner Implementation aber nicht möglich ist. Der Heap wurde über ein Array-Liste implementiert. Dabei handelt es sich um ein Array, das dynamisch erweitert werden kann, wenn der Platz ausgeht. Einen Heap kann man sich als vollständigen, binären Baum vorstellen. Jeder Arrayslot stellt einen Knoten dar, wobei sich die Wurzel am ersten Arrayslot befindet. Die Nachfolger eines Knotens  $i$  haben dann die Indizes  $2i$  und  $2i + 1$ , weil der Baum immer vollständig ist. Für jeden Knoten muss die Heap-Bedingung erfüllt sein, die besagt, dass alle Nachfolger eines Knotens kleiner (oder gleich) dem aktuellen Knoten sein müssen. Beim Einfügen und Entfernen von Knoten muss sichergestellt werden, dass die Heap-Bedingung erfüllt bleibt.

Wenn der erste Knoten in den Heap eingefügt wird, ist die Heap-Bedingung natürlich erfüllt. Für jeden weiteren Knoten wird das folgende Verfahren angewendet. Der neue Knoten wird zunächst ganz rechts unten im Heap eingefügt, also im ersten leeren Arrayslot<sup>51</sup>. Nun kann es sein, dass der eingefügte Knoten größer als sein Vorgänger ist. In diesem Fall ist die Heap-Bedingung verletzt und der Knoten wird so lange mit seinem Vorgänger vertauscht, bis der Knoten an der Wurzel angekommen ist oder der Elternknoten größer als der aktuelle Knoten ist (*Upheap*).

Das Entfernen des Maximums ist etwas komplizierter. Das Maximum befindet sich immer an der Wurzel, deshalb wird diese einfach zurückgegeben. Nun wird die Wurzel gelöscht und das äußerste Element am rechten unteren Rand (das letzte Element im Array) des Baumes tritt an deren Stelle. Dieses Element ist mit sehr großer Wahrscheinlichkeit nicht das maximale Element. Deshalb muss es mit seinen Nachfolgern vertauscht werden. Der Knoten wird so lange mit seinem größeren Kinderknoten vertauscht, bis es keinen größeren Kinderknoten mehr gibt oder der Knoten an den Blättern des Baumes angekommen ist (*Downheap*). Es ist wichtig, dass immer mit dem größeren Kinderknoten vertauscht wird, weil sonst die Heap-Bedingung wieder verletzt werden würde, wenn der kleinere Kinderknoten einen größeren Nachfolger hat. Durch das Entfernen des

---

<sup>49</sup>Die *Dequeue* wurde über eine doppelt verkettete Liste implementiert. Das Ermitteln eines Datensatzes nach seinem Index (Position) ist zwar deshalb nur noch in  $\mathcal{O}(n)$  (anstatt in  $\mathcal{O}(1)$ ) möglich, jedoch wird diese Operation auch nicht benötigt. Einfügen und Löschen (und Ermitteln) des ersten oder letzten Elements sind in konstanter Zeit möglich.

<sup>50</sup>In C# als *default(DequeueItem)* bezeichnet.

<sup>51</sup>Der Heap wird also zeilenweise aufgebaut.

Maximums hat sich die Größe des Heaps um 1 verkleinert.

Die meisten Operationen sind bei einem Heap in logarithmischer Zeit möglich. Für das Einfügen und Entfernen beträgt die Laufzeitkomplexität  $\mathcal{O}(\log_2 n)$ , wenn der Heap  $n$  Elemente enthält, weil ein solcher vollständiger, binärer Baum eine Höhe von  $\log_2 n$  aufweist (Jeder Knoten hat zwei Nachfolger). Ein Knoten kann also maximal  $\log_2 n$  Mal vertauscht werden.

## 2.5 Nutzungsgrenzen

Die meisten Teile des Programms sind gegen falsche Eingaben durch den Benutzer abgesichert. Das Programm sollte also nicht abstürzen, wenn der Benutzer eine falsche Eingabe macht. Nutzungsgrenzen ergeben sich also nur durch die Laufzeitkomplexität. Bei 92 Personen und einer Tischgröße von 300 dauern die einzelnen Schritte der Algorithmen schon ziemlich lange (siehe dazu Programm-Ablaufprotokoll), etwa 10 Sekunden. Wenn die Tischgröße geändert werden soll, muss das Programm neu gestartet werden. Ein Tisch kann erzeugt werden, indem auf den blauen Link im Hauptfenster geklickt wird. Theoretisch könnte man mehrere Tische erzeugen, es wird aber immer nur der erste Tisch beachtet, da ich es aus verschiedenen Gründen nicht für sinnvoll hielt, eine Erweiterung für mehrere Tische zu entwickeln<sup>52</sup>. Im Menü Spielregeln kann man Einstellungen für den Koordinator vornehmen (Anzahl der freien Personen usw.). Will man normale Personen hinzufügen, wählt man im Menü Restaurant/Erweiterungen den Menüpunkt Normale Personen. Mit dem Slider am unteren Fensterrand kann man die Geschwindigkeit der Simulation verändern. Die Geschwindigkeit kann nur *vor* dem Start der Simulation verändert werden.

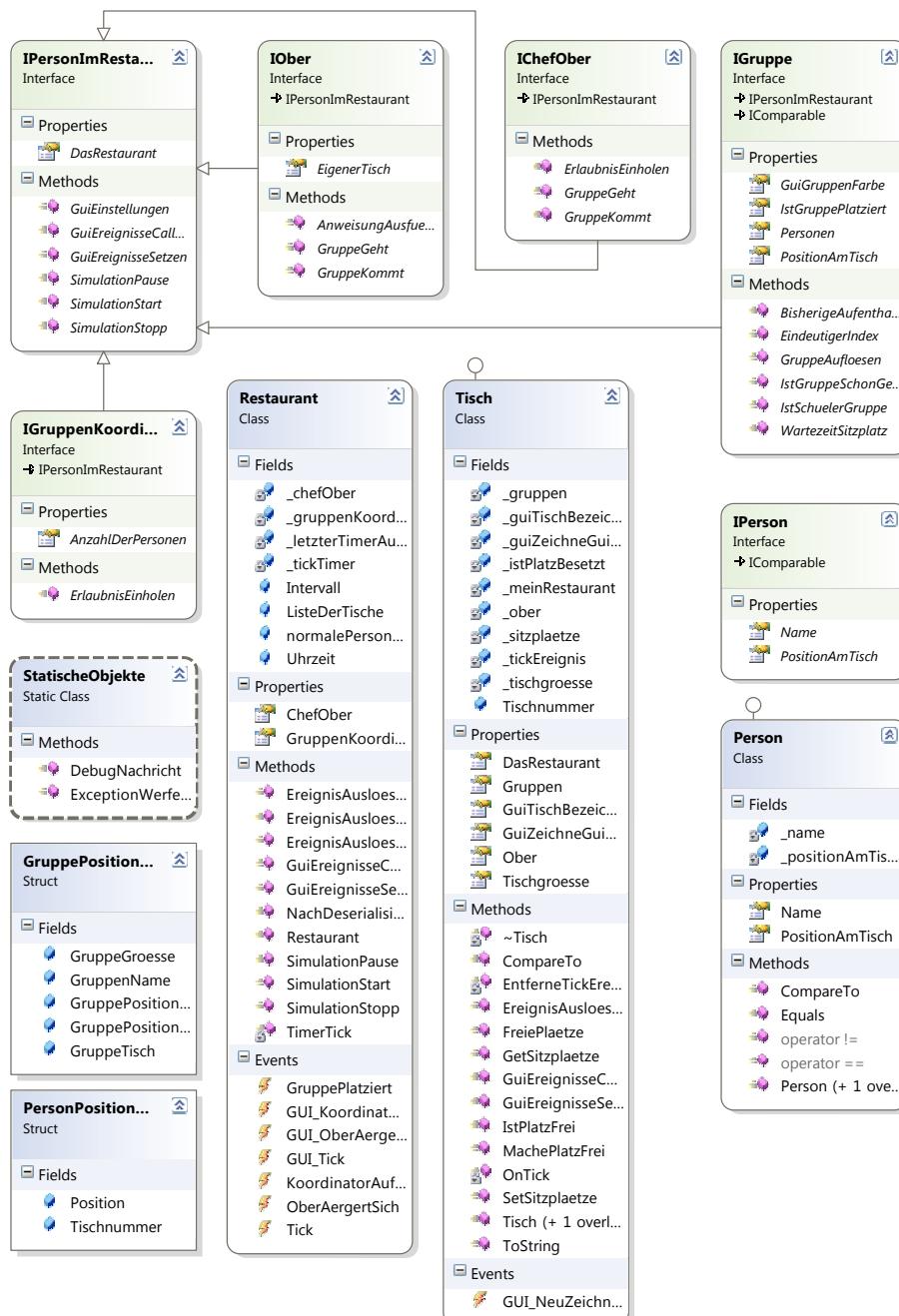
**Hinweis** Wenn die Simulation gestartet wird, ohne dass ein einziger Tisch hinzugefügt wurde, stürzt das Programm ab.

## 2.6 Klassendiagramme

Zum Abschluss der Programm-Dokumentation folgen nun noch ein paar Klassendiagramme, die einen groben Überblick über das Programm verschaffen sollten. Besonders die Bedeutung der einzelnen Klassen und Interfaces sollte dadurch klar werden, falls der Leser nicht nur an den Algorithmen, sondern auch am Aufbau des Programms interessiert ist.

---

<sup>52</sup>Diese Erkenntnis kam mir natürlich erst später...



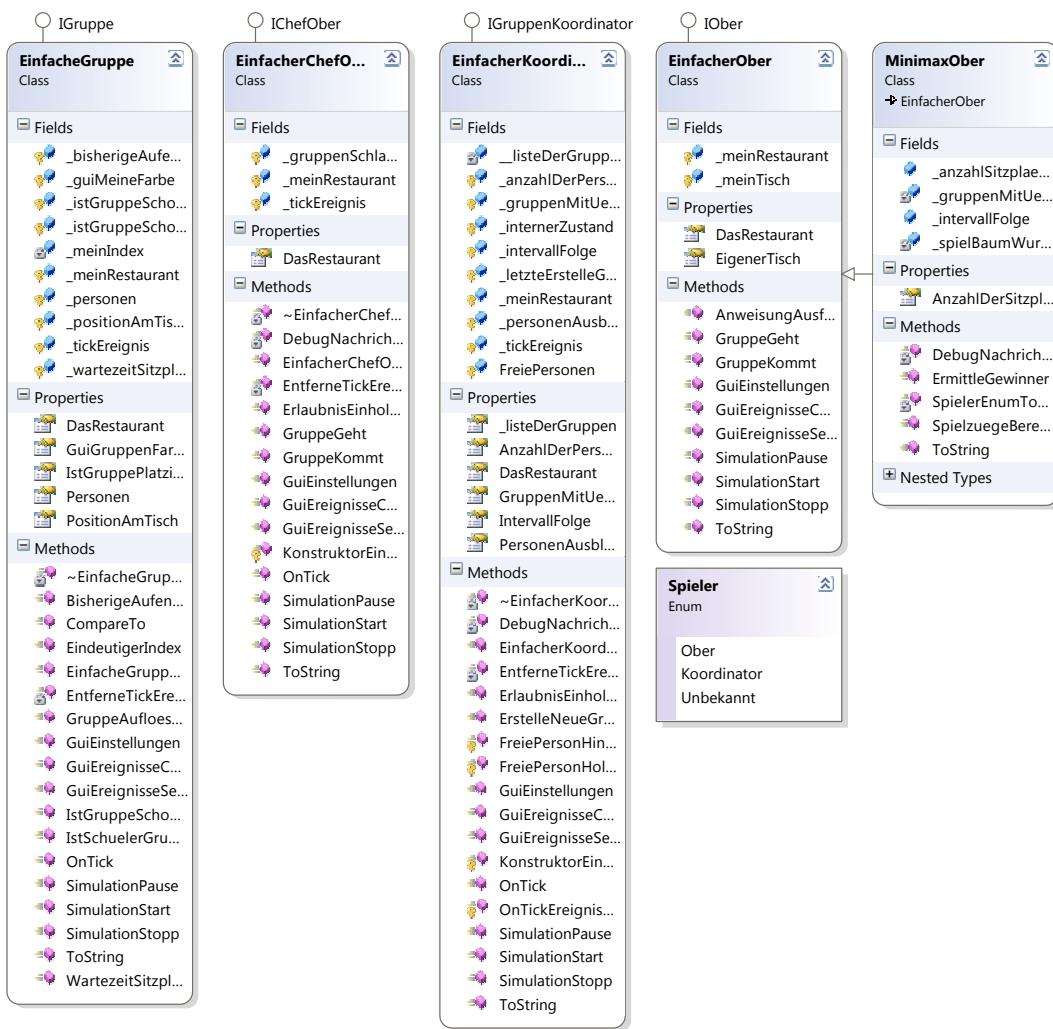
Interfaces.cs – Erläuterungen zum Klassendiagramm

<b>IPersonImRestaurant</b>	Dieses Interface müssen Ober, ChefOber, Gruppe und Koordinator implementieren. Es wird für die GUI benötigt, damit für jedes Objekt ein Eigenschaftsfenster erzeugt werden kann. Außerdem kann so jedes Objekt darüber informiert werden, dass die Simulation begonnen hat.
<b>void GuiEinstellungen()</b>	Gibt ein Eigenschaftsfenster zurück, vom Typ SteuerelementEinstellungen, das vom UserControl erbt.
<b>GuiEreignisseCallBackZeigerLoeschen()</b> und <b>GuiEreignisseSetzen()</b>	Wird nicht mehr benötigt.
<b>void SimulationStart()</b>	Wird aufgerufen, wenn die Simulation beginnt. Das Restaurant-Objekt legt dabei eine komplette Kopie der Klasse mit allen Unterklassen an (Serialisierung).
<b>void SimulationPause()</b>	Wird aufgerufen, wenn die Simulation pausiert wird (nicht implementiert).
<b>void SimulationStopp()</b>	Wird aufgerufen, wenn die Simulation gestoppt wird. Das zuvor angelegte Restaurant-Objekt wird deserialisiert, sodass sich das Programm wieder im Anfangszustand befindet.
<b>IOber</b>	Dieses Interface muss der einfache Ober implementieren.
<b>EigenerTisch</b>	Zeiger auf den Tisch des Obers.
<b>void AnweisungAusfuehren (...)</b>	Der Chefober kann den Ober anweisen, eine Anweisung auszuführen. Wird nicht benötigt (war für Erweiterungen gedacht).
<b>void GruppeGeht (<i>IGruppe</i> gruppe)</b>	Wird aufgerufen, wenn eine Gruppe geht. Die Sitzplätze am Sitzplatzarray werden freigegeben.
<b>GruppePositionAmTisch</b> <b>GruppeKommt (<i>IGruppe</i> gruppe)</b>	Wird aufgerufen, wenn eine Gruppe kommt. Der Ober darf die Gruppe platzieren.
<b>IChefOber</b>	Der einfache Chefober muss dieses Interface implementieren.
<b>bool ErlaubnisEinholen(...)</b>	Der Ober fragt den Chefober, ob die Aktion, die er durchführen will (Gruppe platzieren etc.) so erlaubt ist. Wird nicht verwendet und war für Erweiterungen gedacht.
<b>void GruppeGeht (<i>IGruppe</i> gruppe)</b>	Wird aufgerufen, wenn eine Gruppe geht. Der Chefober leitet das an den zuständigen Ober weiter (nur sinnvoll bei mehreren Tischen, für Erweiterung gedacht).
<b>GruppePositionAmTisch</b> <b>GruppeKommt (<i>IGruppe</i> gruppe)</b>	Wird aufgerufen, wenn eine Gruppe kommt. Der Chefober fügt die Gruppe in die Dequeue ein und gibt die Anweisung dann an den Ober

	weiter.
<b>IGruppe</b>	Jede Gruppe muss dieses Interface implementieren (einfache Gruppe und normale Gruppe)
<code>System.Drawing.Color GuiGruppenFarbe</code>	Legt die Farbe fest, die die Gruppe in der GUI hat.
<code>List&lt;IPerson&gt; Personen</code>	Eine Liste mit allen Personen, die die Gruppe enthält.
<code>bool IstGruppePlatziert</code>	Legt fest, ob die Gruppe schon vom Ober platziert wurde.
<code>GruppePositionAmTisch PositionAmTisch</code>	Gibt die Position der Gruppe im Restaurant (Tisch, Platzintervall) zurück.
<code>int EindeutigerIndex()</code>	Gibt den eindeutigen Index der Gruppe zurück.
<code>bool IstGruppeSchonGegangen()</code>	Legt fest, ob die Gruppe schon gegangen ist. Wenn das der Fall ist, kann die Gruppe gelöscht werden.
<code>bool IstSchuelerGruppe()</code>	Legt fest, ob es sich um eine Schülergruppe oder um eine normale Gruppe handelt.
<code>int BisherigeAufenthaltsdauer()</code>	Gibt zurück, wie lange die Gruppe schon im Restaurant ist.
<code>int WartezeitSitzplatz()</code>	Legt fest, wie lange die Gruppe noch wartet, bis sie nach Hause geht (hier war eine Erweiterung geplant), weil sie nicht platziert wurde.
<code>void GruppeAufloesen()</code>	Löst die Gruppe auf.
<b>IGruppenKoordinator</b>	Der einfache Gruppenkoordinator muss dieses Interface implementieren.
<code>int AnzahlDerPersonen</code>	Legt die Anzahl der freien Personen fest.
<code>bool ErlaubnisEinholen(...)</code>	Wird nicht benötigt (für Erweiterung gedacht).
<b>IPerson</b>	Dieses Interface muss jede Person implementieren (Person und PersonGeld).
<code>PersonPositionAmTisch PositionAmTisch</code>	Gibt die Position der Person am Tisch zurück.
<code>string Name</code>	Nicht benötigt (für Erweiterung gedacht).
<b>Restaurant</b>	Die zentrale Restaurant-Klasse.
<code>private object _chefOber</code>	Zeiger auf den Chefober.
<code>private object _gruppenKoordinator</code>	Zeiger auf den Gruppenkoordinator.
<code>public List&lt;Tisch&gt; ListeDerTische</code>	Eine Liste mit allen Tischen im Restaurant.
<code>public int Intervall</code>	Legt die Simulationsgeschwindigkeit fest (Intervall des Timers).
<code>public int Uhrzeit</code>	Gibt die aktuelle Uhrzeit zurück.
<code>public NormalePersonen</code>	Ein Zeiger auf die Klasse, die für das Einfügen der normalen Personen zuständig ist.
<code>private bool _letzterTimerAufrufSchonFertig</code>	Diese Variable ist notwendig, damit nicht ein neuer OnTick-Aufruf gestartet wird, wenn der Vorherige noch nicht fertig ist. Sonst könnte z.B. der Ober schon zum Zug kommen, während der Koordinator noch die einzelnen Schritte berechnet.

<code>private Timer _tickTimer</code>	Der Timer, der die OnTick-Aufrufe auslöst.
[Verschiedene Ereignis auslösen-Funktionen]	Sind notwendig, damit Chefober und Koordinator Ereignisse des Restaurants auslösen können.
<code>GuiEreignisseCallBackZeigerLoeschen und GuiEreignisseSetzen</code>	Nicht mehr benötigt.
<code>public void NachDeserialisierung</code>	Erstellt ein neues Timer-Objekt, da dies nicht serialisiert werden kann. Sonst würden nämlich auch alle Delegates des Tick-Ereignisse serialisiert werden und damit die gesamte GUI.
<code>public Restaurant()</code>	Der Konstruktor initialisiert die Liste der Tische und das Objekt <code>normalePersonen</code> .
<code>private void TimerTick(...)</code>	Wird aufgerufen, wenn das Tick-Ereignis des Timers ausgelöst wird. Die Uhrzeit wird erhöht und das Tick-Ereignis des Restaurants wird ausgelöst.
<b>Tisch</b>	Repräsentiert einen Tisch.
<code>private int _tischgroesse</code>	Anzahl der Sitzplätze.
<code>private Restaurant _meinRestaurant</code>	Zeiger auf das Restaurant.
<code>private string _guiTischBezeichner</code>	Name des Tisches.
<code>private bool _guiZeichneGuibeijedemTick</code>	Legt fest, wie oft die GUI den Tisch neu zeichnen soll.
<code>public int Tischnummer</code>	Der Index des Tisches.
<code>private List&lt;IGruppe&gt;</code>	Eine Liste aller Gruppen am Tisch.
<code>private IOber _ober</code>	Zeiger auf den zuständigen Ober.
<code>private IGruppe[] _sitzplaetze</code>	Das Sitzplatzarray. Jedem Sitzplatz wird eine Gruppe zugewiesen. Auf welchem Platz welche Person sitzt kann nicht bestimmt werden.
<code>private TickEreignis _tickEreignis</code>	Ein Funktionszeiger auf das Tick-Ereignis beim Restaurant.
<code>private bool[] _istPlatzBesetzt</code>	Legt fest, ob es Platz besetzt ist.
<code>public Tisch()</code>	Der Konstruktor des Tisches.
<code>~Tisch()</code>	Der Destruktur des Tisches, der das Tick-Ereignis beim Restaurant abmeldet.
<code>public int CompareTo(object obj)</code>	Nicht benötigt. War für eine Erweiterung gedacht. Ein Tisch ist größer, wenn er mehr freie Plätze hat.
<code>public int FreiePlaetze()</code>	Gibt die Anzahl der freien Plätze am Tisch zurück.
<code>public bool IstPlatzFrei(int platznummer)</code>	Gibt „wahr“ zurück, wenn ein Platz frei ist.
<code>public void MachePlatzFrei(int platznummer)</code>	Setzt eine Platz auf „frei“.
<code>private void OnTick(long uhrzeit)</code>	Weist die GUI an, den Tisch neu zu zeichnen.
<code>public void SetSitzplaetze(int index, IGruppe gruppe)</code>	Besetzt einen Platz.
<code>public Tisch(Restaurant restaurant)</code>	Konstruktor des Tisches, der die Ereignisse beim Restaurant automatisch abfängt.

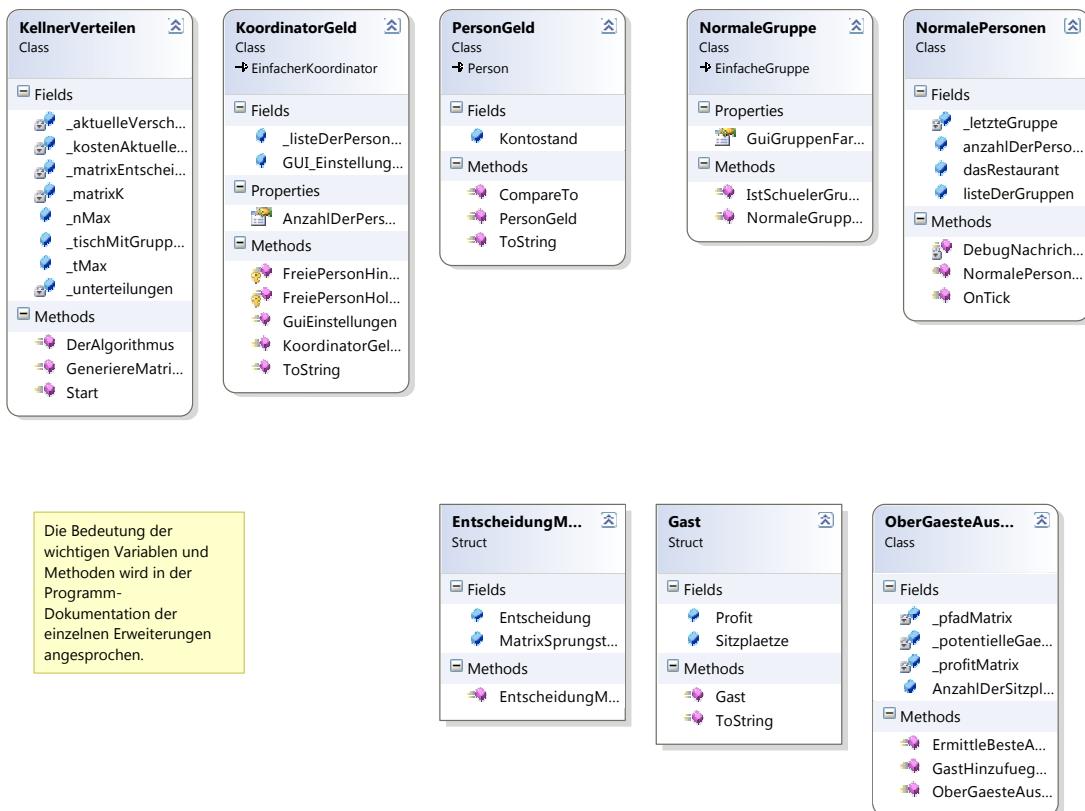
<b>GruppePositionAmTisch</b>	Legt die Position einer Gruppe fest.
<code>public string GruppenName</code>	Name der Gruppe (nicht verwendet).
<code>public int GruppePositionStart</code>	Startindex am Tisch.
<code>public int GruppePositionEnde</code>	Index des letzten Platzes am Tisch.
<code>public int GruppeGroesse</code>	Anzahl der Personen.
<code>public Tisch GruppeTisch</code>	Zeiger auf den Tisch.
<b>StatischeObjekte</b>	Statische Klasse für Debugzwecke



### EinfachesRestaurant.cs und MinimaxRestaurantOber.cs – Erläuterungen zum Klassendiagramm

Das Klassendiagramm zeigt alle wichtigen Klassen für das einfache Restaurant. Dabei wurde zuvor schon darauf eingegangen, was die einzelnen Methoden/Funktion/Eigenschaften dieser Klassen zu bedeuten haben. Der MinimaxOber erbt nur deshalb vom einfachen Ober, weil ich mir so einige Zeilen Quellcode für Tisch- und Restaurant-Variablen sparen konnte.

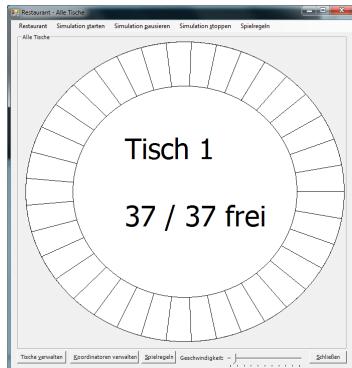
<b>MinimaxOber</b>	Enthält den perfekten Ober.
<code>public int[] _intervallFolge</code>	Speichert die Intervallfolge für den einfachen Koordinator, der simuliert werden soll.
<code>private bool _gruppenMitUebrigenPersonenAuffuellen</code>	Eine Option des einfache Koordinators.
<code>public int AnzahlDerSitzplaetze</code>	Legt die Größe des Tisches fest.
<code>public ZustandGewinner ErmittleGewinner(SpielbaumKnoten zustand, Spieler aktuellerSpieler, int rekursionstiefe)</code>	Ermittelt den Gewinnen für einen bestimmten Zustand. Es folgen weitere rekursive Aufrufe.
<code>public List&lt;string&gt; SpielzuegeBerechnen(int freiePersonen)</code>	Diese Funktion wird von der GUI aufgerufen, für eine bestimmte Anzahl an freien Personen wird berechnet, wer gewinnt. Diese Funktion bereitet das Programm vor (Variablen initialisieren usw.) und startet dann die Funktion ErmittleGewinner.
<code>private string SpielerEnumToString(Spieler spieler)</code>	Nur für Debugzwecke.
<code>private void DebugNachrichtAusgeben(string text, int rekursionstiefe)</code>	Nur für Debugzwecke.



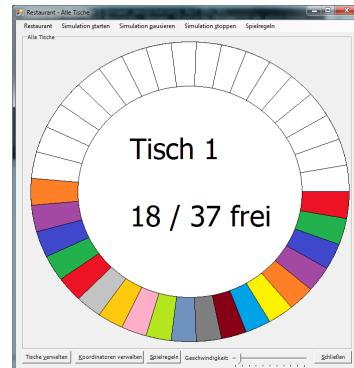
### 3 Programm-Ablaufprotokoll

#### 3.1 Einfacher Ober versus Einfacher Koordinator

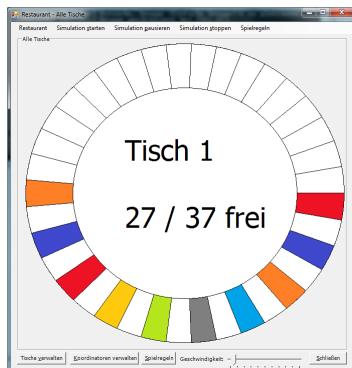
Tischgröße: 37, Anzahl der freien Personen: 19.  
Einfacher Ober ärgert sich.



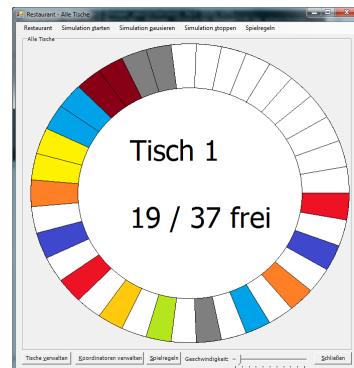
Der einfache Ober platziert die 19 Gruppen (je 1 Person) direkt nebeneinander, um keine Lücken entstehen zu lassen (platzsparend).



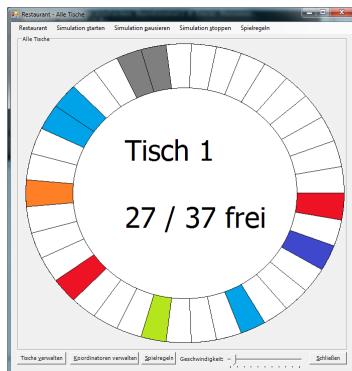
Der einfache Koordinator erzeugt Lücken der Größe 1, dabei werden 9 Personen frei.



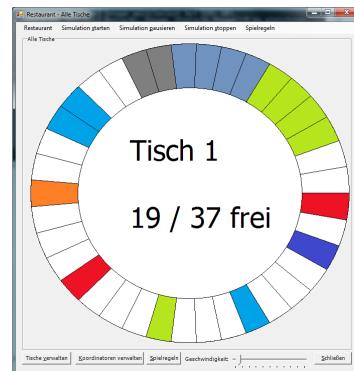
Es werden 4 Gruppen (je 2 Personen) erstellt und platziert, eine Person bleibt übrig. Es wird wieder platzsparend platziert.



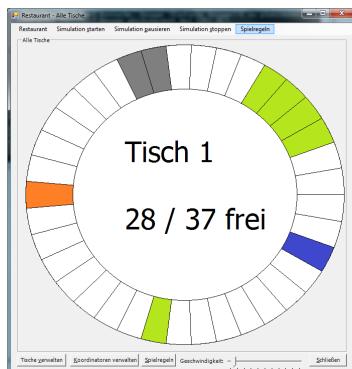
Der einfache Koordinator erzeugt Lücken der Größe 3, dabei werden 8 Personen frei (insgesamt 9 frei).



Der einfache Ober platziert die 2 Gruppen (je 4 Personen) direkt nebeneinander, um keine Lücken entstehen zu lassen (platzsparend), eine Person bleibt übrig.



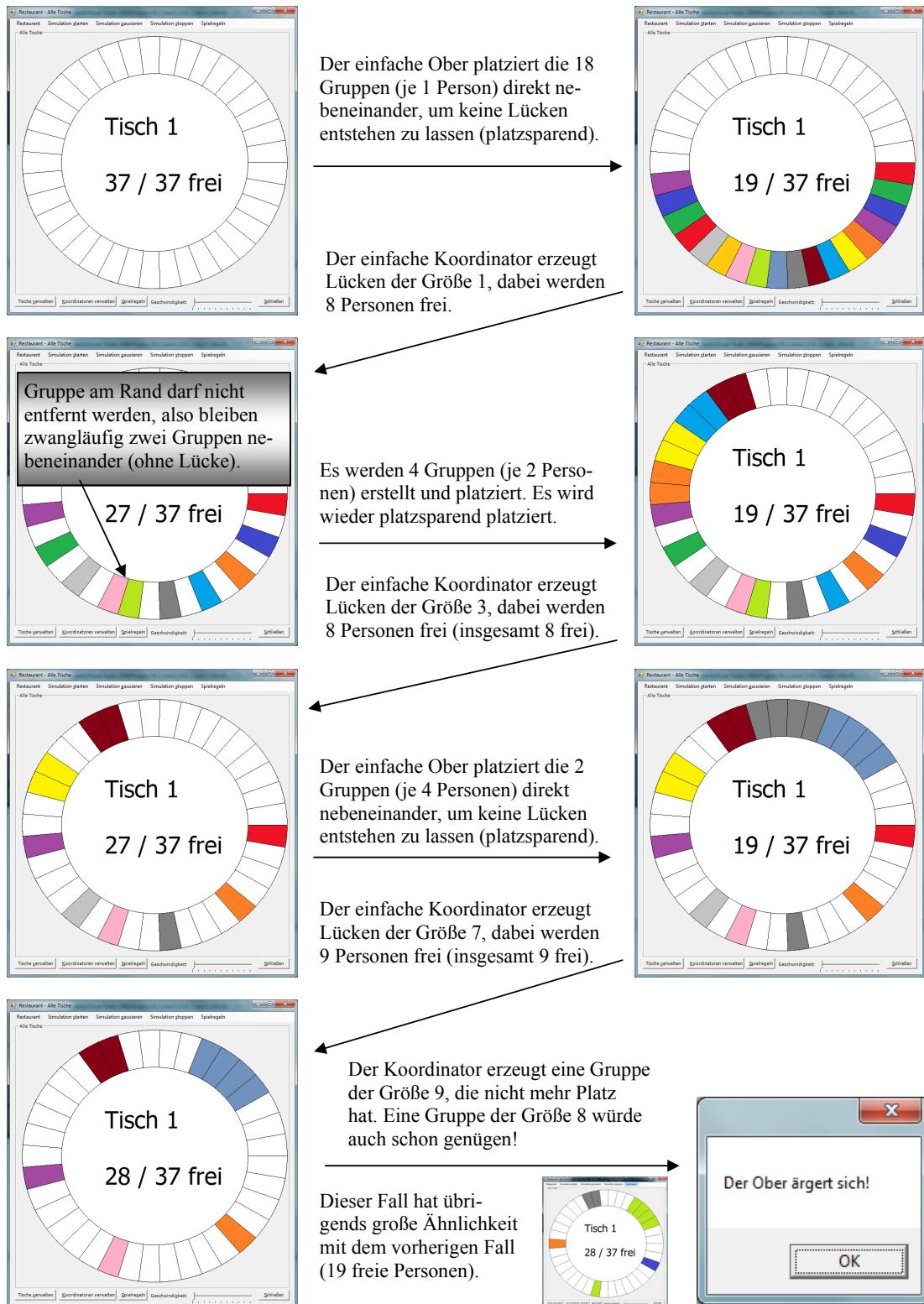
Der einfache Koordinator erzeugt Lücken der Größe 7, dabei werden 9 Personen frei (insgesamt 10 frei).



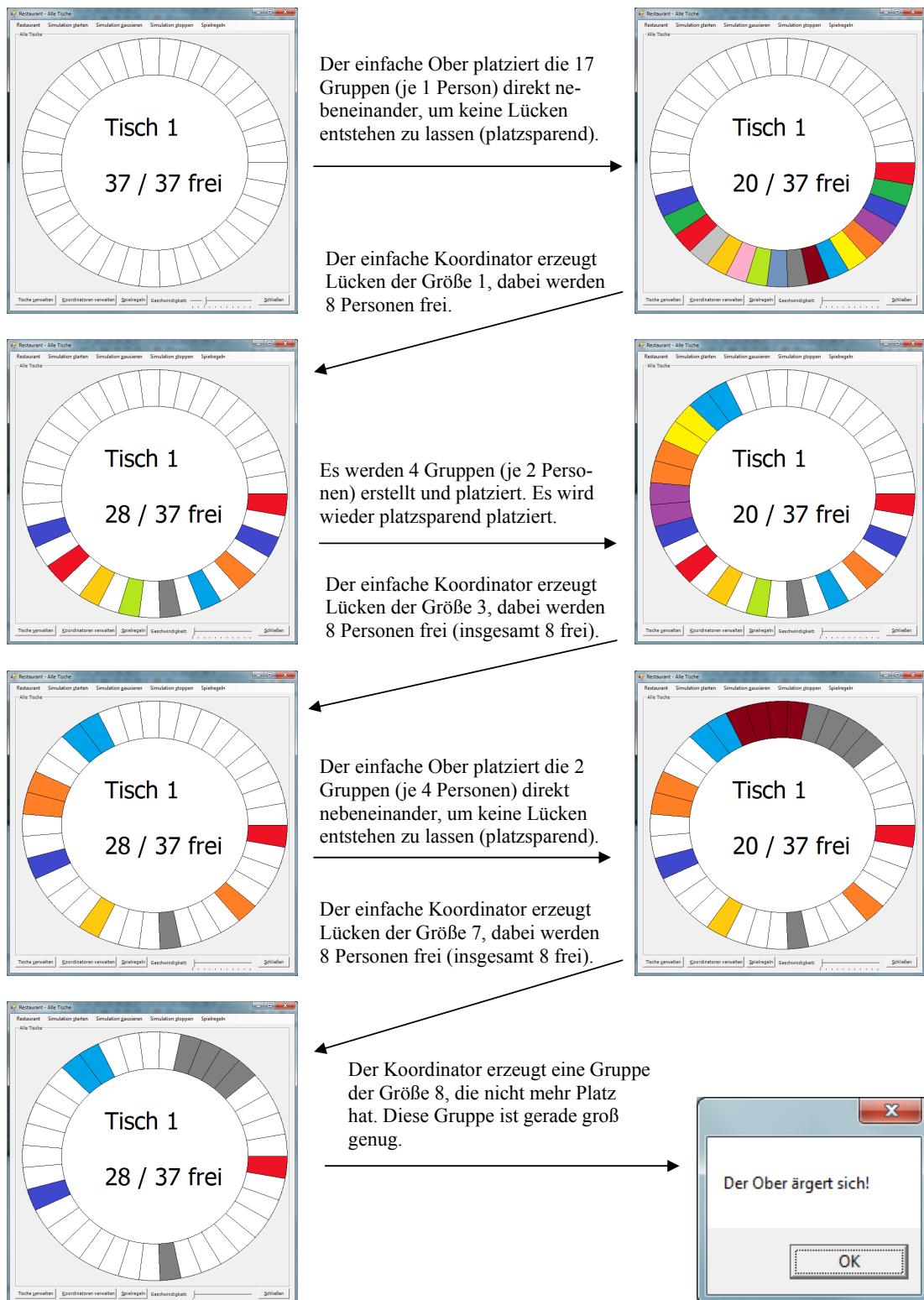
Der Koordinator erzeugt eine Gruppe der Größe 10, die nicht mehr Platz hat. Eine Gruppe der Größe 8 würde auch schon genügen!



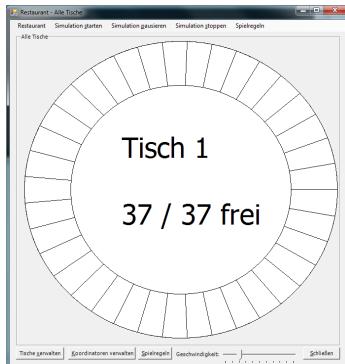
Tischgröße: 37, Anzahl der freien Personen: 18.  
Einfacher Ober ärgert sich.



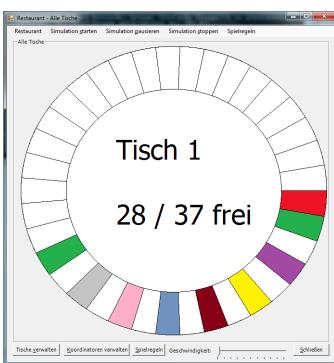
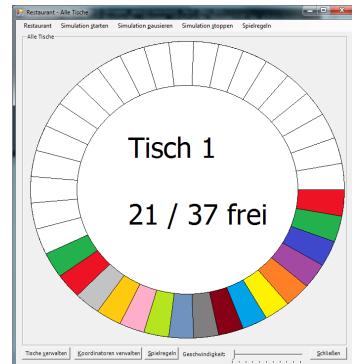
Tischgröße: 37, Anzahl der freien Personen: 17.  
Einfacher Ober ärgert sich.



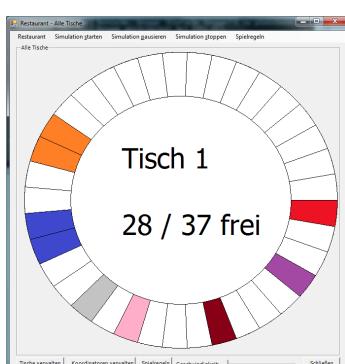
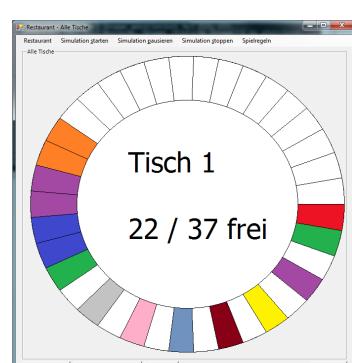
Tischgröße: 37, Anzahl der freien Personen: 16.  
Koordinator gibt auf.



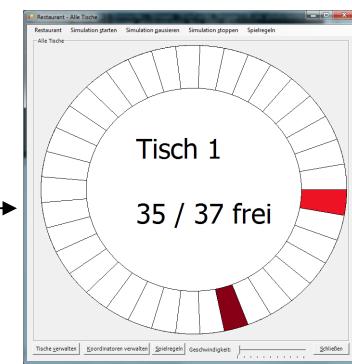
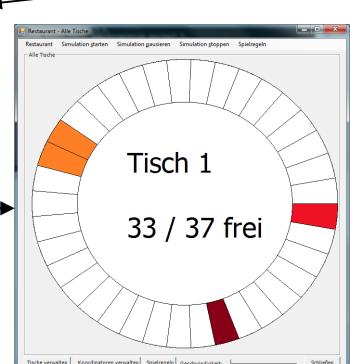
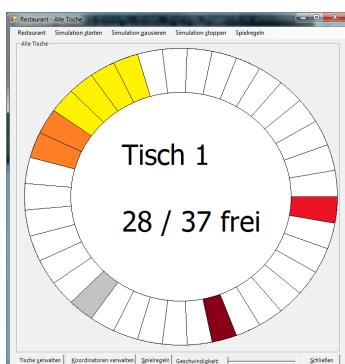
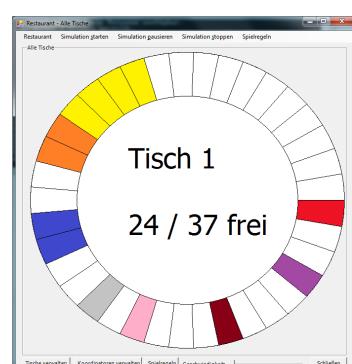
Wenn man diesen Fall mit den vorherigen Fällen vergleicht, fallen große Ähnlichkeiten auf. Der gravierende Unterschied ist, dass im siebten Bild die größte Lücke 11 Plätze groß ist. Der Algorithmus hat zuvor Lücken mit einer Größe von 7 erstellt. Die 11er-Lücke war aber schon vorher zu groß.



Außerdem stehen zu diesem Zeitpunkt lediglich 7 freie Personen zur Verfügung. Der Algorithmus möchte eigentlich 8er-Gruppen erstellen, was aber nicht möglich ist (da nur 7 freie Personen). Deshalb entfernt er die nächsten Gruppen, dieses Mal mit 15er-Lücken.

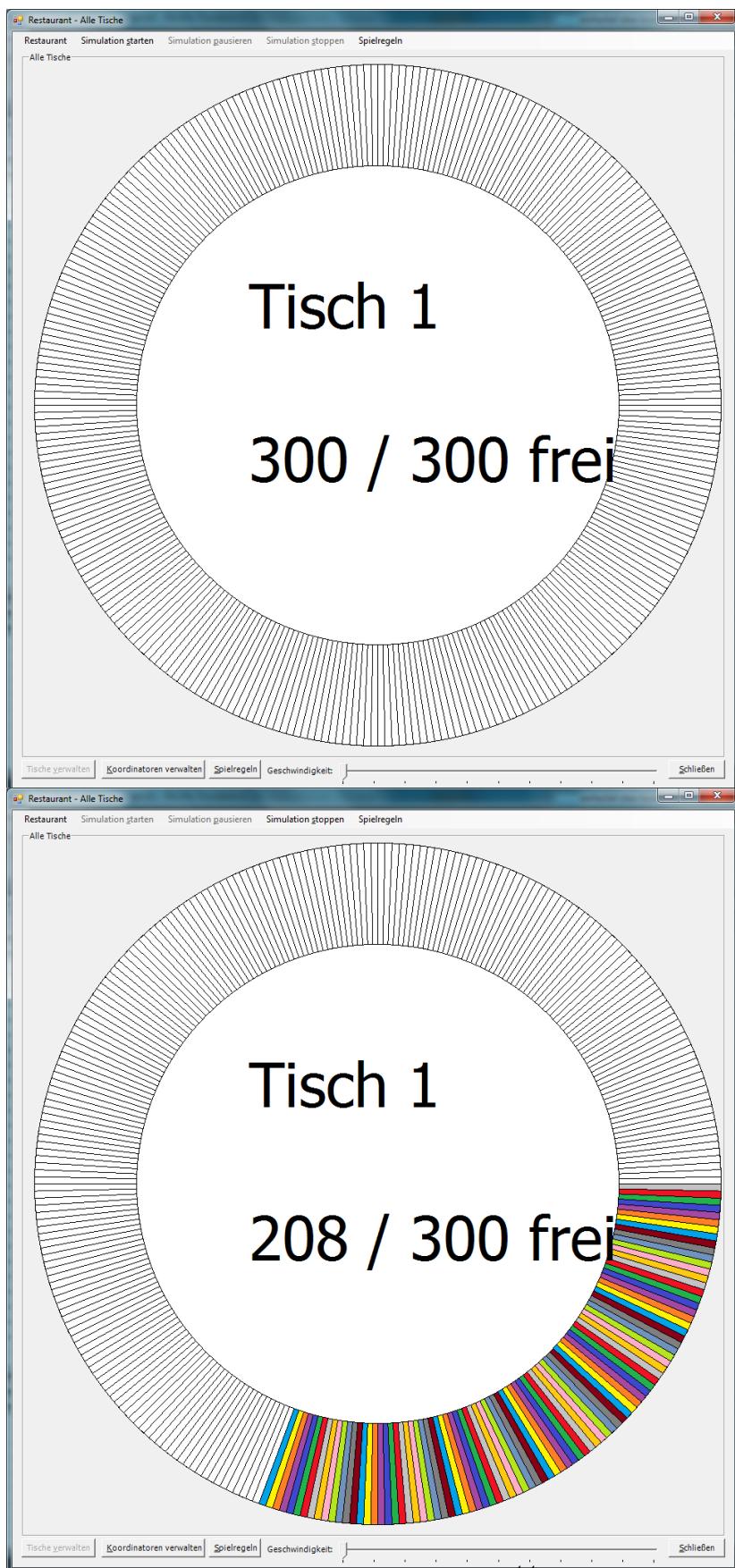


Dadurch werden aber immer noch nicht genug Personen frei, also geht es weiter mit 31er-Lücken. Danach ist Schluss, weil einerseits die Folge nur bis 64 geht und andererseits 63 bereits größer als die Anzahl der Plätze ist. Der Koordinator gibt auf. Es sind mindestens 17 Personen notwendig!



Das folgende Beispiel soll zeigen, dass das Programm auch mit großen Problemen gut umgehen kann. Noch größere wären zwar auch noch möglich, allerdings wäre dann auf der Oberfläche nichts mehr zu erkennen. Außerdem steigt die Laufzeit mit zunehmender Problemgröße. Bei dem folgenden Problem mit 300 Sitzplätzen und 92 freien Personen dauert der Vorgang schon gefühlte zehn Sekunden. Das hat meiner Meinung nach die folgenden Gründe.

1. Es dauert einige Zeit, die GUI (den Tisch) zu zeichnen. Je größer der Tisch ist, desto länger dauert der Vorgang.
2. Auch wenn die Laufzeitkomplexität auf dem Papier vielleicht gut aussieht, könnte man den Algorithmus (über die Konstanten) noch viel schneller machen. Es werden häufig Objekte umgewandelt, Datenstrukturen kopiert usw. Hier könnte man bestimmt noch einiges verbessern. Ich vermute, dass man das Programm um den Faktor 0,5 beschleunigen könnte, wenn man komplett auf die komplexen Klassen und Strukturen verzichtet und stattdessen nur primitive Datentypen verwendet.



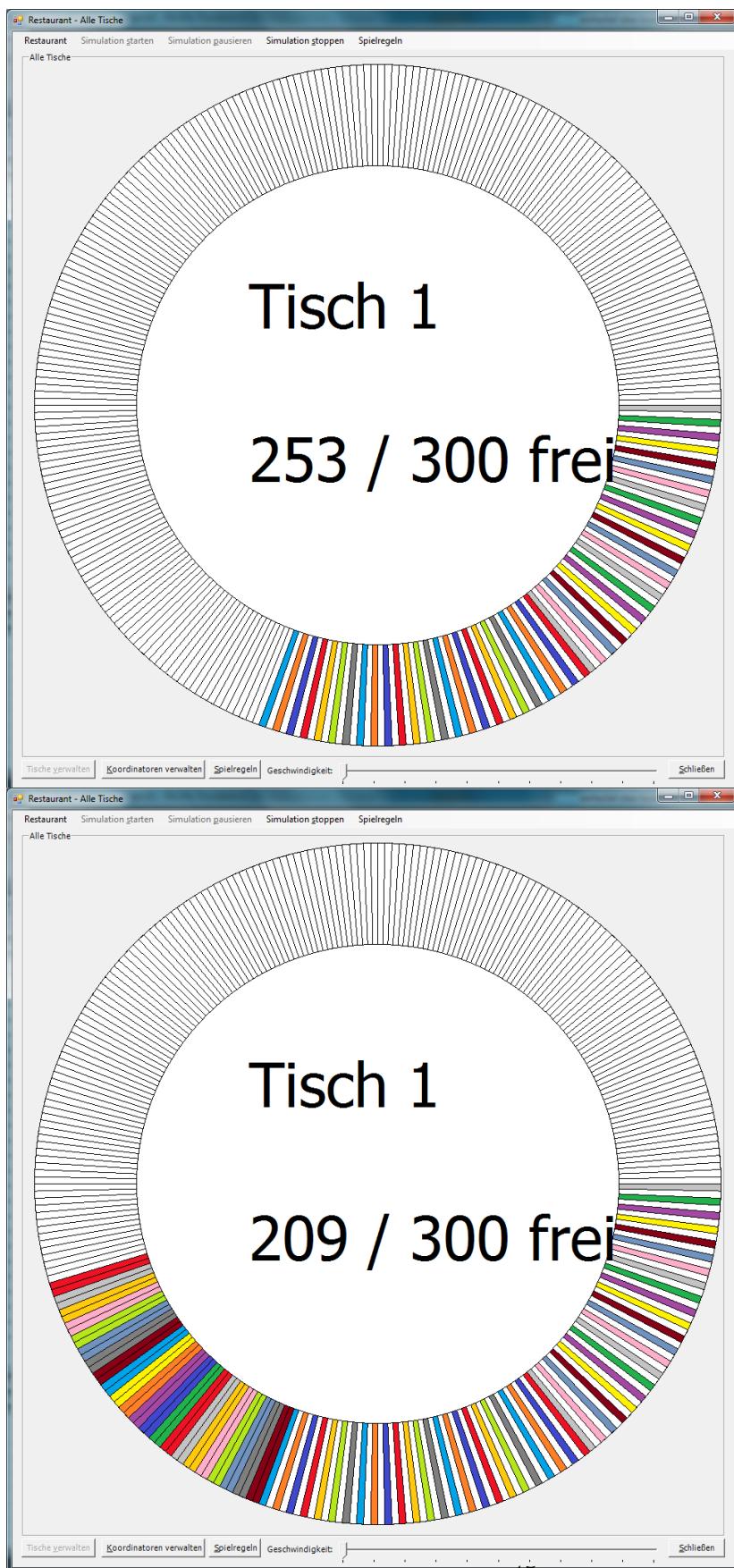
## Einfacher Ober vs. Einfacher Koordinator

Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen

Der Tisch ist anfangs noch komplett leer.

Es werden 92 Gruppen aus je einer Person erstellt. Diese Gruppen werden platzsparend nebeneinander platziert.



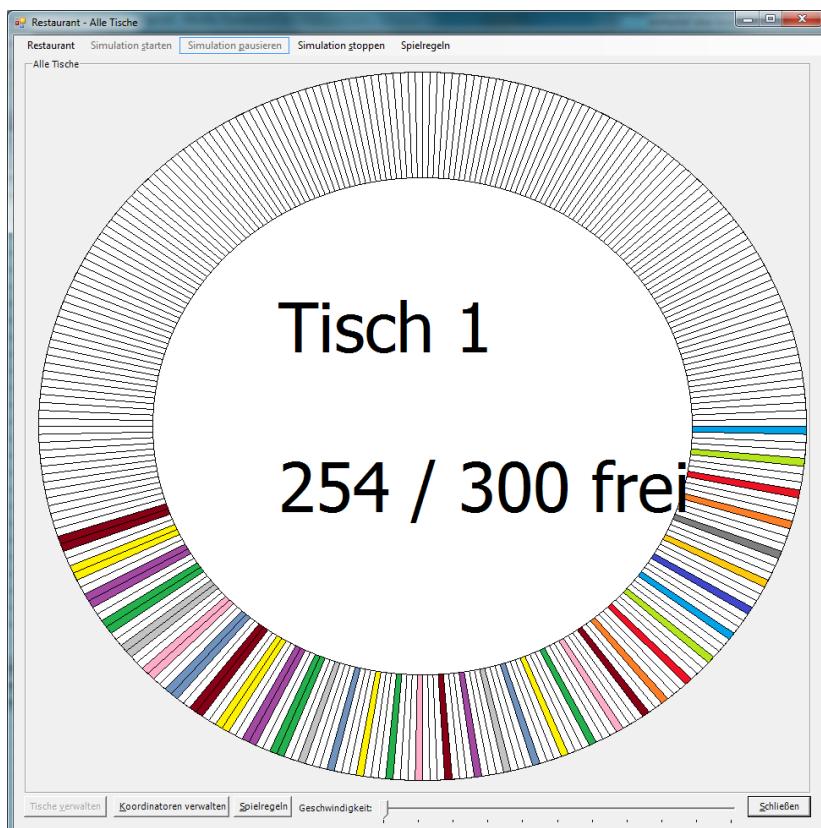
## Einfacher Ober vs. Einfacher Koordinator

Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen
- Folge (2,4,8,16,32,64,128)

Es werden 45 Personen/Gruppen entfernt, sodass Lücken der Größe 1 entstehen.

Aus den 45 freien Personen können 22 Gruppen mit je 2 Personen erstellt werden. Eine Person bleibt übrig.

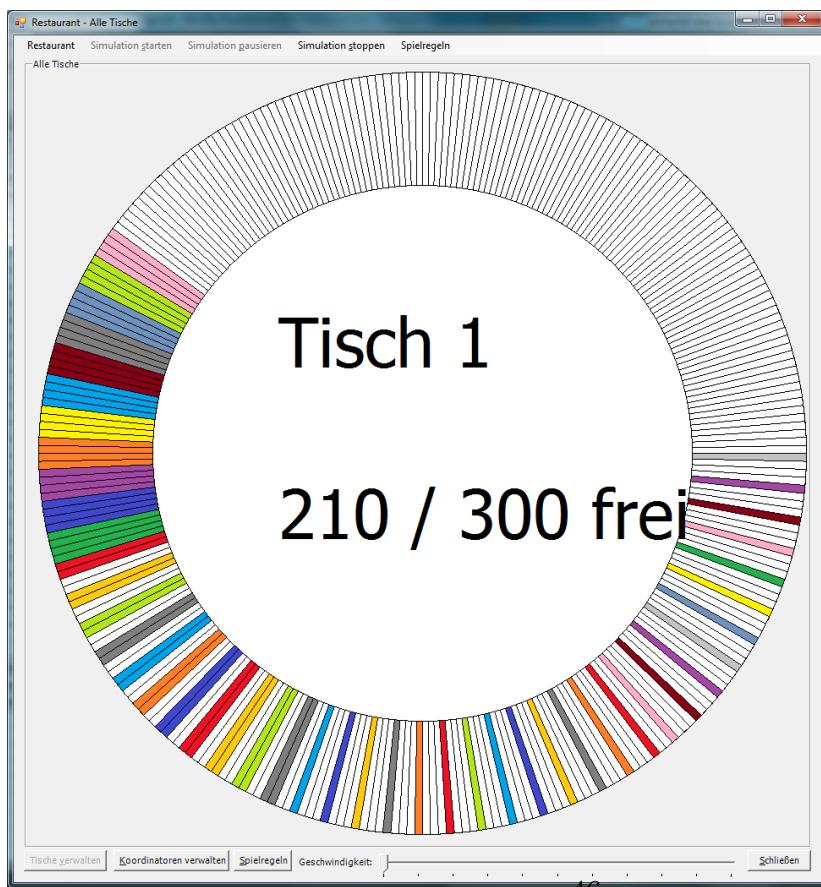


### Einfacher Ober vs. Einfacher Koordinator

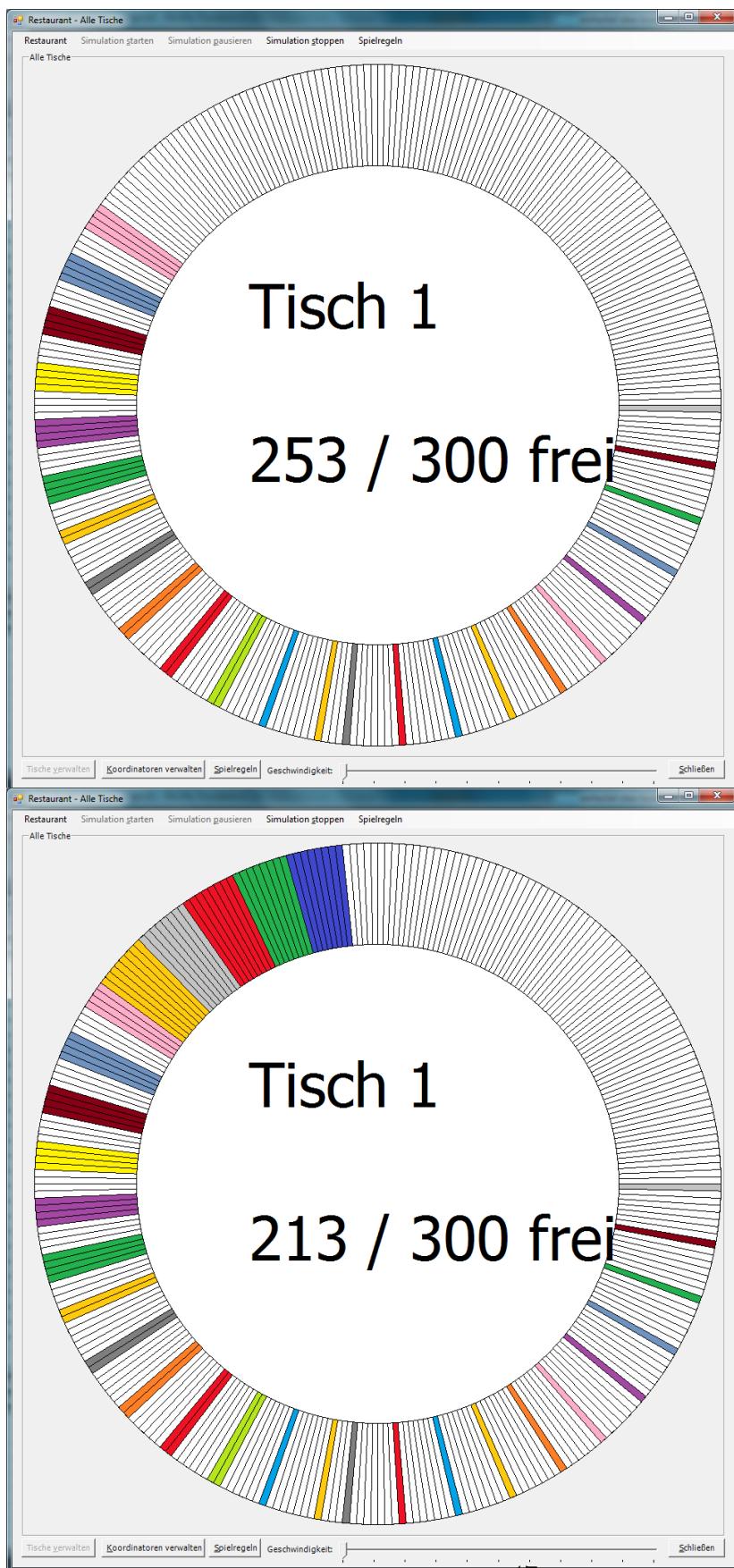
Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen
- Folge (2,4,8,16,32,64,128)

Es werden 45 Personen entfernt, sodass Lücken der maximalen Größe 3 entstehen.  
Insgesamt stehen nun 46 freie Personen zur Verfügung.



Aus den 46 freien Personen können 11 Gruppen der Größe 4 erstellt werden. Dabei bleiben 2 freie Personen übrig.



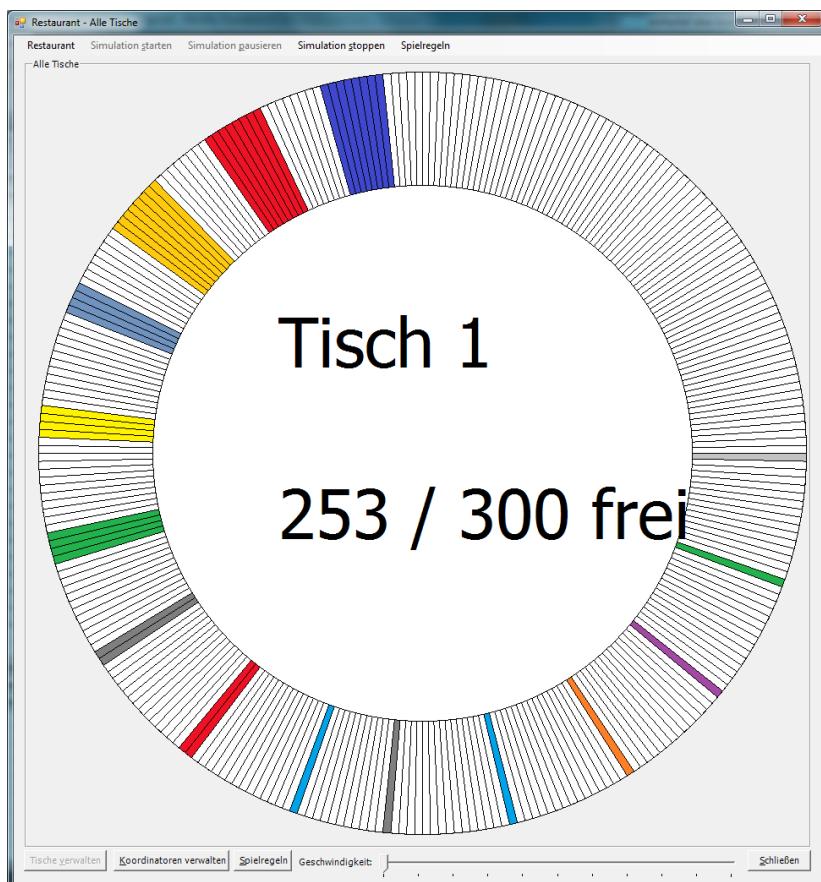
## Einfacher Ober vs. Einfacher Koordinator

Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen
- Folge (2,4,8,16,32,64,128)

Es werden 43 Personen entfernt, sodass Lücken der maximalen Größe 7 entstehen.  
Insgesamt stehen nun 45 freie Personen zur Verfügung.

Aus den 45 freien Personen können 5 Gruppen der Größe 8 erstellt werden. Dabei bleiben 5 freie Personen übrig.

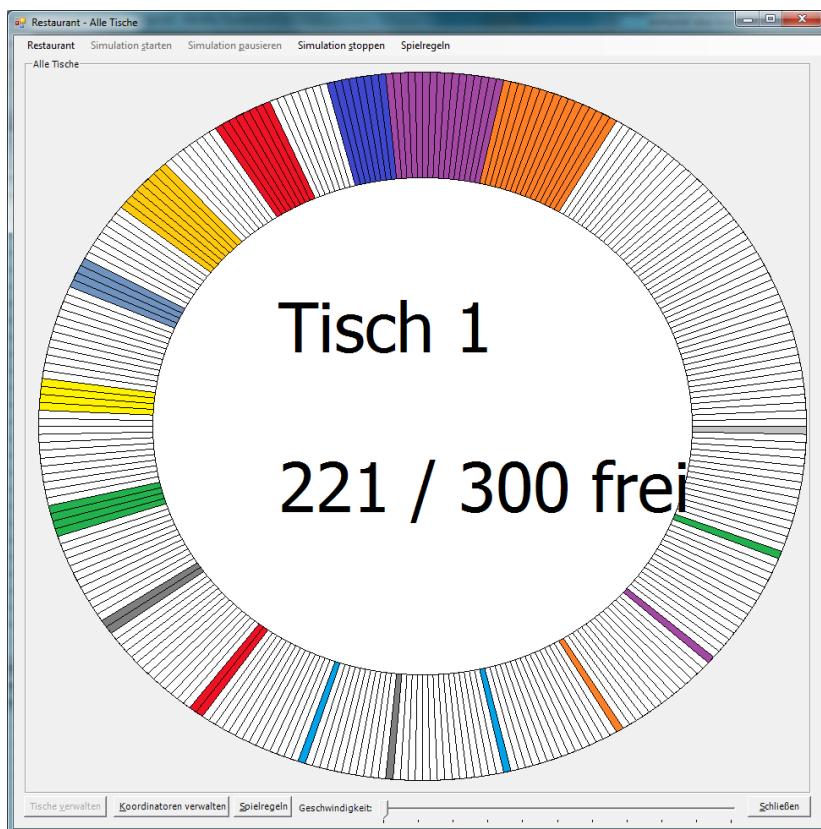


### Einfacher Ober vs. Einfacher Koordinator

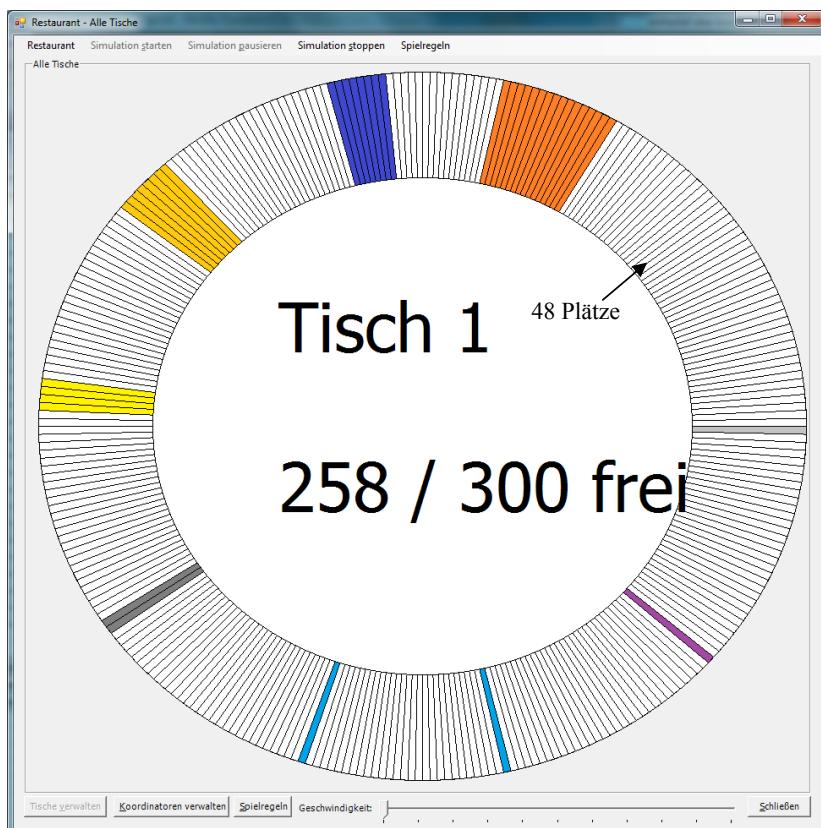
Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen
- Folge (2,4,8,16,32,64,128)

Es werden 40 Personen entfernt, sodass Lücken der maximalen Größe 15 entstehen. Insgesamt stehen nun 45 freie Personen zur Verfügung.



Aus den 45 freien Personen können 2 Gruppen der Größe 16 erstellt werden. Dabei bleiben 13 freie Personen übrig.

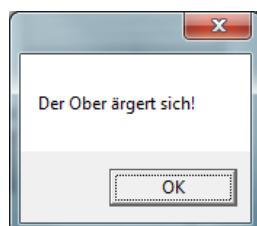


### Einfacher Ober vs. Einfacher Koordinator

Für diesen Fall gilt:

- 300 Sitzplätze
- 92 freie Personen
- Folge (2,4,8,16,32,64,128)

Es werden 37 Personen entfernt, sodass Lücken der maximalen Größe 31 entstehen. Insgesamt stehen 50 freie Personen zur Verfügung.



Die größte Lücke ist 48 Plätze groß (hoffentlich hab ich mich nicht verzählt...). Es stehen 50 Personen zur Verfügung. Eine Gruppe der Größe 50 kann also nicht mehr untergebracht werden.

### 3.2 Einfacher Ober versus Einfacher Koordinator - Ein Überblick

Die folgende Tabelle enthält eine Übersicht darüber, wie viele freie Personen bei einer bestimmten Tischgröße benötigt werden, damit sich der Ober ärgert. Dabei spielt der einfache Koordinator gegen einen einfachen Ober und einen perfekten Ober. Jede Spalte steht für eine Tischgröße. Befindet sich hinter der Zahl noch der Buchstabe  $P$ , so spielt der Koordinator gegen den perfekten Ober. Die Zeilen stehen für die Anzahl der freien Personen (ganz zu Beginn). Jeder Eintrag  $K$  in der Tabelle bedeutet, dass der Koordinator gewinnt.  $O$  bedeutet, dass der Ober gewinnt. Wenn eine Zelle ein Fragezeichen enthält, bedeutet das, dass die Berechnung zu lange dauerte und abgebrochen wurde. Wenn es eine Strategie gibt, sodass der Ober gewinnt, findet der perfekt Ober diese in der Regel relativ schnell. Wenn irgendeine solche Strategie gefunden wurde, kann nämlich abgebrochen werden. Falls es keine solche Strategie gibt, müssen alle möglichen Spielzüge untersucht werden. Während der Algorithmus des perfekten Obers innerhalb von wenigen Sekunden eine Lösung für das Problem  $25P/13$  gefunden hatte, lief die Berechnung für den Fall  $25P/14$  eine viertel Stunde, bevor ich den Vorgang abgebrochen habe. Man kann mit relativ großer Sicherheit davon ausgehen, dass der einfache Koordinator in diesem Fall gewinnt.

Die untere Tabelle zeigt weitere Ergebnisse des einfachen Obers. Es wird immer die Anzahl an Personen abgegeben, die notwendig ist, damit der Ober sich ärgert.

Gewinner	Tisch	5	5P	6	6P	7	7P	8	8P	9	9P	10	10P	15	15P	20	20P	25	25P
Personen																			
1		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	
2		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	
3		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	
4		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	
5		K	K	K	K	O	O	O	O	O	O	O	O	O	O	O	O	O	
6		K	K	K	K	K	K	O	O	O	O	O	O	O	O	O	O	O	
7		K	K	K	K	K	K	K	K	K	K	O	O	O	O	O	O	O	
8		K	K	K	K	K	K	K	K	K	K	O	O	O	O	O	O	O	
9								K	K	K	K	O	O	O	O	O	O	O	
10								K	K	K	K	O	O	O	O	O	O	O	
11								K	K	K	K	O	O	O	O	O	O	O	
12								K	K	K	K	O	O	O	O	O	O	O	
13								K	K	K	K	O	O	O	O	O	O	O	
14								K	K	K	K	?	?	?	?	?	?	?	
15								K	K	K	K	?	?	?	?	?	?	?	
16								K	K	K	K	?	?	?	?	?	?	?	
17								K	K	K	K	?	?	?	?	?	?	?	
18								K	K	K	K	?	?	?	?	?	?	?	
19								K	K	K	K	?	?	?	?	?	?	?	
20								K	K	K	K	?	?	?	?	?	?	?	
21								K	K	K	K	?	?	?	?	?	?	?	
22								K	K	K	K	?	?	?	?	?	?	?	
23								K	K	K	K	?	?	?	?	?	?	?	
24								K	K	K	K	?	?	?	?	?	?	?	
25								K	K	K	K	?	?	?	?	?	?	?	
Quote		1	1	0,83	0,83	0,86	0,86	0,75	0,88	0,78	0,78	0,7	0,7	0,6	0,67	0,55	0,6	0,52	?

Koordinator gewinnt mit Folge  
(2, 3, 8, 16, 32, 64)

↓

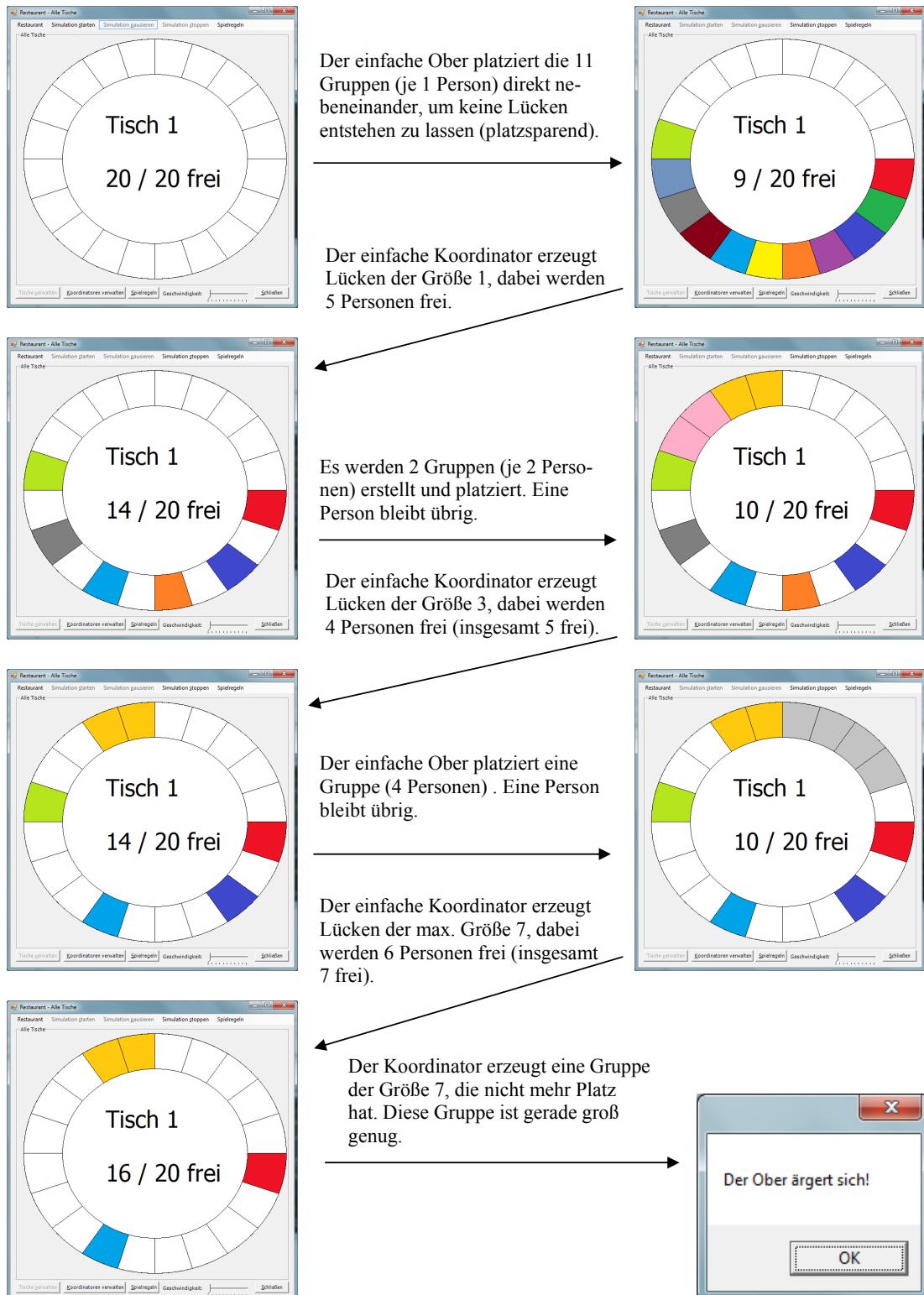
Intervallfolge 2, 4, 8, 16, 32, 64

### 3.3 Vergleich: Einfacher Ober - Perfekter Ober

Die folgenden zwei Seiten zeigen den Fall *20 Sitzplätze* und *11 freie Personen*. Der einfache Ober ärgert sich, während der perfekte Ober eine Strategie kennt, wie er sich nicht ärgern muss. Dazu können nun die einzelnen Schritte verglichen werden.

Den entscheidenden Unterschied findet man beim perfekten Ober im vierten Schritt. Die zwei Gruppen der Größe 2 werden dort nicht platzsparend platziert. Stattdessen wird eine große Lücke vom Ober zerstückelt. Allerdings geschieht das so, dass der Koordinator offenbar Probleme hat, die neuen Lücken effizient zu nutzen.

Tischgröße: 20, Anzahl der freien Personen: 11.  
Einfacher Ober ärgert sich.



**Einfacher Koordinator vs. Perfekter Ober: 20P/11**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Anfangs ist das Restaurant komplett leer. Der Koordinator erzeugt 11 Gruppen der Größe 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1	2	3	4	5	6	7	8	9	10	11	0	0	0	0	0	0	0	0	0

Der perfekte Ober platziert die Gruppen so, wie der einfache Ober sie platzieren würde.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	1	0	3	0	5	0	7	0	9	0	11	0	0	0	0	0	0	0	0	

Der einfache Koordinator löscht Gruppen, sodass Lücken der Größe 1 entstehen. Dabei werden fünf Personen frei. Es werden zwei Gruppen mit je zwei Personen erstellt.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4	1	0	3	0	5	0	7	0	9	0	11	12	12	0	0	0	0	13	13	0

Die neuen zwei Gruppen platziert der perfekt Ober nicht platzsparend. Stattdessen wird die große Lücke in der Mitte zerstückelt.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
5	0	0	3	0	0	0	7	0	9	0	0	12	12	0	0	0	0	13	13	0

Der einfache Koordinator löscht Gruppen, sodass Lücken mit einer maximalen Größe von drei entstehen. Dabei werden drei Personen frei. Insgesamt sind es also vier Personen.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
6	0	0	3	0	0	0	7	0	9	0	0	12	12	14	14	14	14	13	13	0

Die Gruppe der Größe Vier passt nur in eine Lücke.

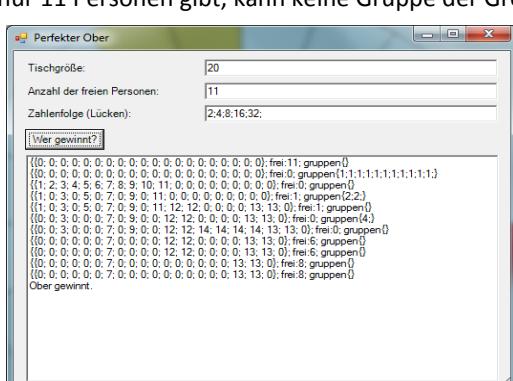
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
7	0	0	0	0	0	0	7	0	0	0	0	12	12	0	0	0	0	13	13	0

Es werden nun Gruppen entfernt, sodass Lücken mit einer maximalen Größe von 7 entstehen.

Dabei werden 6 Personen frei. Es kann keine Gruppe der Größe 8 erstellt werden.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
8	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	13	13

Es wurden weitere Gruppen entfernt, sodass Lücken der Größe 15 entstehen. Da es insgesamt nur 11 Personen gibt, kann keine Gruppe der Größe 16 mehr erstellt werden.



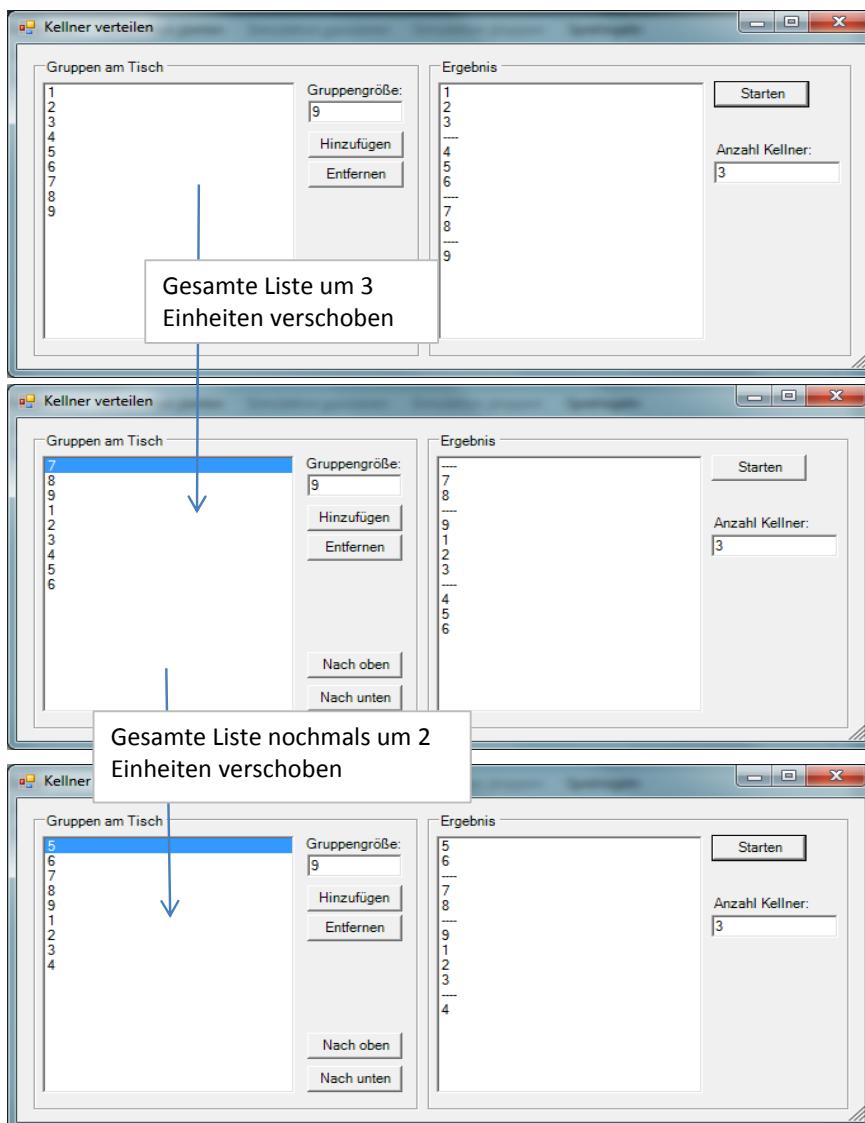
Folge: 2, 4, 8, 16, 32

Anzahl der freien Personen: 11

Tischgröße: 20

### 3.4 Zuständigkeit einzelner Kellner

Es folgen nun ein paar Beispiele, die zeigen sollen, dass der Algorithmus korrekt funktioniert. Dabei ist zu beachten, dass es sich um einen kreisförmigen Tisch handelt. Die Listenbox sieht ein bisschen so aus, als würde es sich um einen rechteckigen Tisch handeln. Außerdem handelt es sich bei den Einträgen in der Listenbox um Gruppengrößen und nicht um Gruppenindizes oder sonst irgendetwas. Leere Plätze am Tisch können ignoriert werden. Lücken dürfen auch zertrennt werden, denn es ist in der Regel sowieso notwendig, die Gruppen neu unter den Kellnern aufzuteilen, wenn neue Personen hinzukommen.

**9 Gruppen, 3 Kellner**

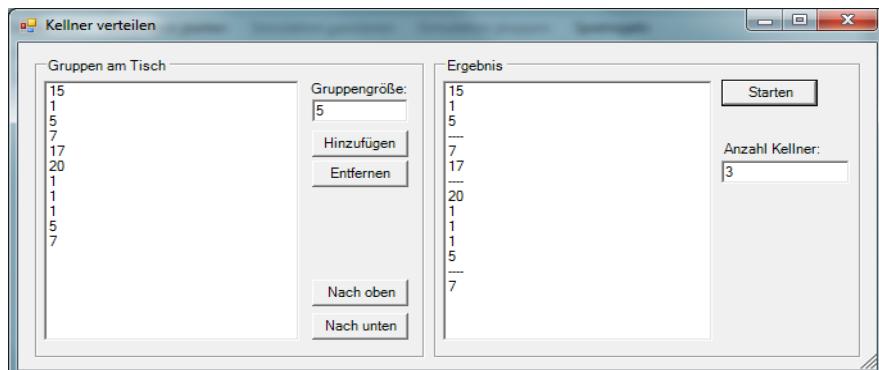
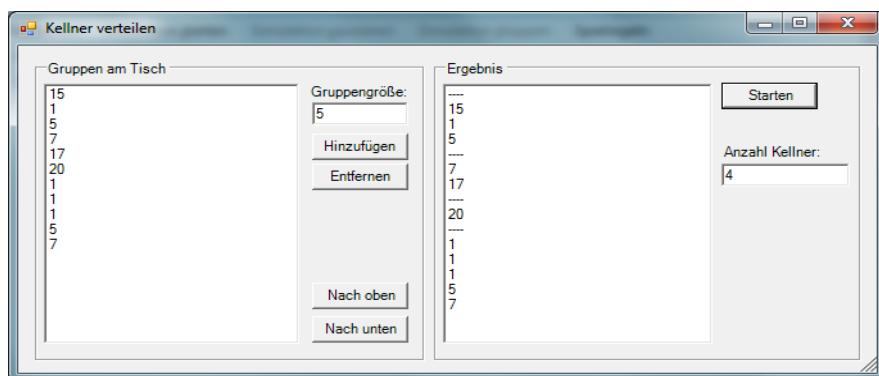
Wie man sieht, macht es nichts, wenn man die ganze Liste um mehrere Einheiten verschiebt. Es kommt immer die gleiche (optimale) Lösung heraus. Der Teil des Algorithmus, der alle  $n_{\max}$  Möglichkeiten generiert, um die letzte Teilung optimal zu setzen, funktioniert also!

Hinweis: Man beachte, dass es sich um einen runden Tisch handelt. Im letzten Beispiel kommt also nach der Gruppe der Größe 4 wieder die Gruppe der Größe 5.

Wie man sieht, ist die Lösung optimal: Alle Segmente haben die gleiche Anzahl an Personen: 15 Personen.

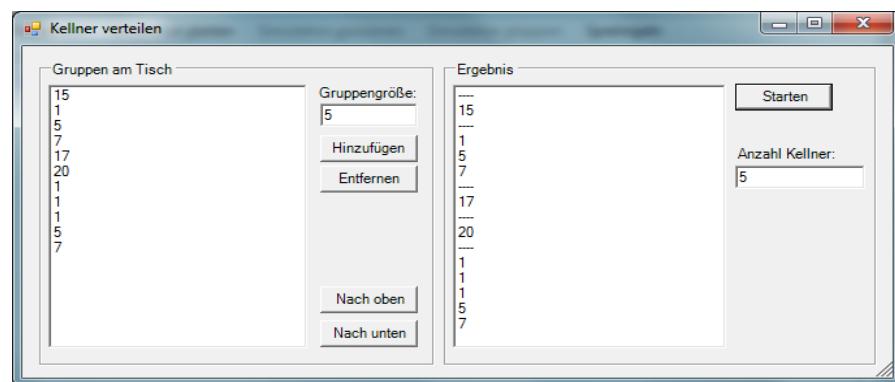
**Einige weitere Beispiele mit 11 Personen**

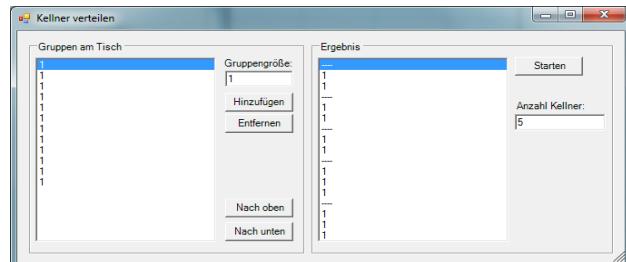
Personen im Segment	
28	
24	
28	
21	
24	
20	
15	

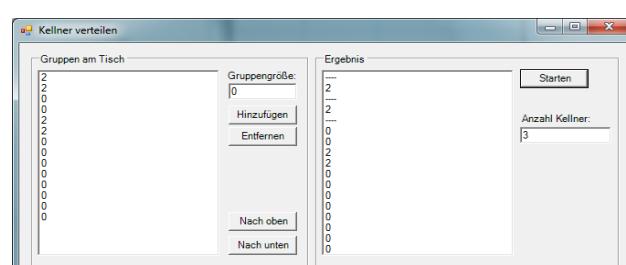
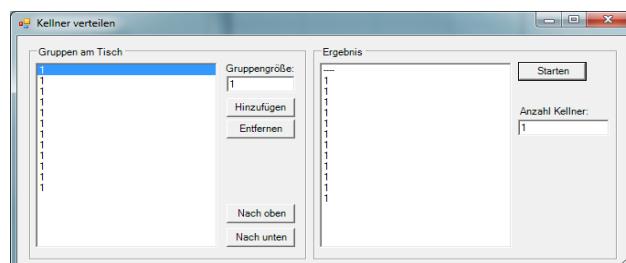
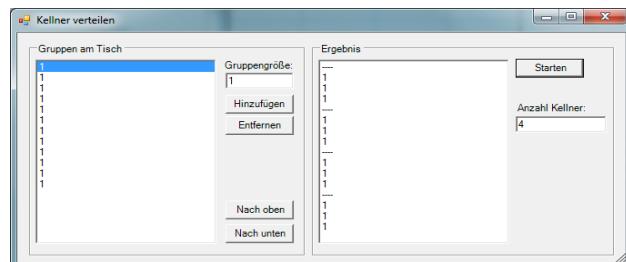
Das letzte Segment hat nur 15 Personen. Würde man die darüber liegende Gruppe noch hinzufügen, so hätte man 35 Personen und würde man die "darunter" liegende Gruppe noch hinzufügen, hätte man 30 Personen. Scheinbar geht es wirklich nicht besser.

15	
13	
17	
20	
15	

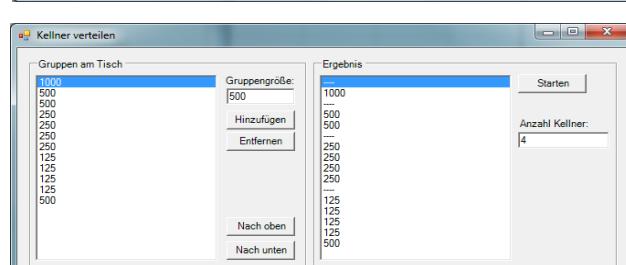


**Einige triviale Beispielefälle**

Wie erwartet:  
Die Gruppen der Größe  
1 wurden so gut wie  
möglich aufgeteilt.

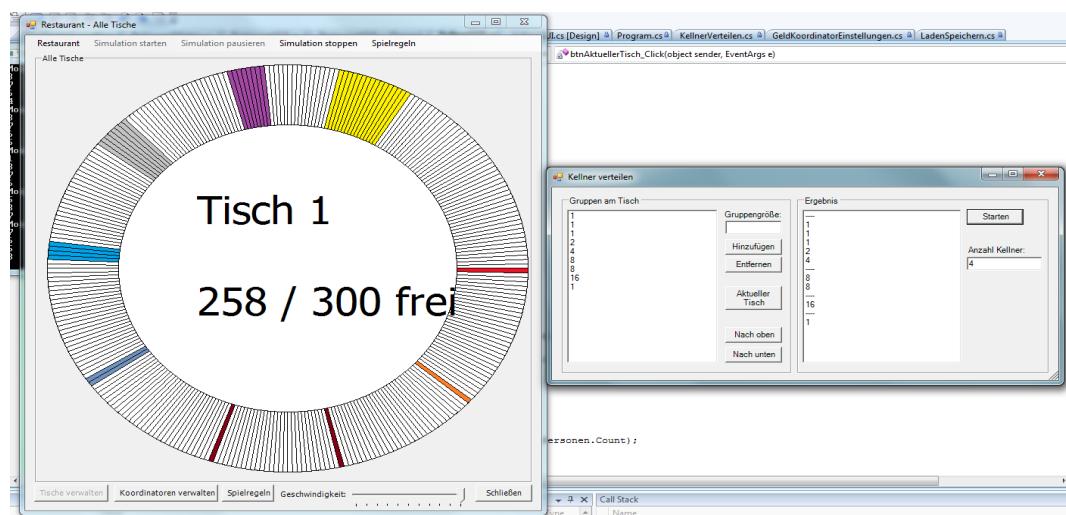


Gruppen der Größe 0  
(auch wenn es die  
formal nicht gibt)  
erzeugen natürlich  
keine Kosten.

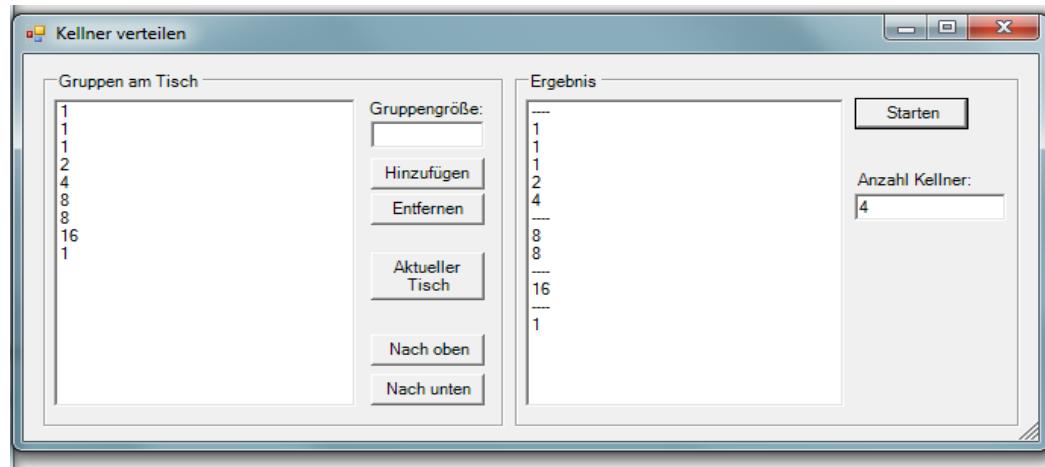


Gleich große Segmente.

**Kellner verteilen: Beispielfall für Anzahl der Plätze: 300, 92 freie Personen**



Auf Wunsch überträgt das Programm den kompletten Tisch automatisch in die Listenboxen.



Es handelt sich hierbei übrigens um einen sehr interessanten Fall. Eigentlich wäre es sinnvoller, dass das letzte Tischsegment mehr als nur eine Person enthält. Man würde eigentlich eine Trennung zwischen den Einsen und der Zwei (oder an einer ähnlichen Stelle) erwarten. Dadurch würde sich jedoch die Größe des größten Segmentes (16 Personen) nicht verringern. Der Fall wird dadurch auch nicht "fairer" es gibt immer noch einen Kellner, der 16 Personen zu bedienen hat.  
Der Algorithmus versucht schließlich, die Größe des größten Segmentes zu verringern und teilt den Tisch dadurch nur indirekt fair auf.

### 3.5 Reservierungen im Turmrestaurant

Die folgende Tabelle zeigt einige Beispielausgaben der Erweiterung. In den ersten beiden Zeilen sind alle Gruppen aufgelistet, wobei zu jeder Gruppe angegeben ist, wie groß sie ist und welchen Profit sie verursacht. Aus den folgenden Zeilen wird ersichtlich, welche Gruppen bei einer bestimmten Tischgröße ausgewählt werden.

Gruppen mit einem sehr guten Profit-Personen-Verhältnis (wenig Personen, viel Profit) werden besonders häufig ausgewählt. Das ist zum Beispiel bei den Gruppen 2-5 und 17-50 der Fall.

## Sitzplatzreservierungen: Einige Beispiele Fälle

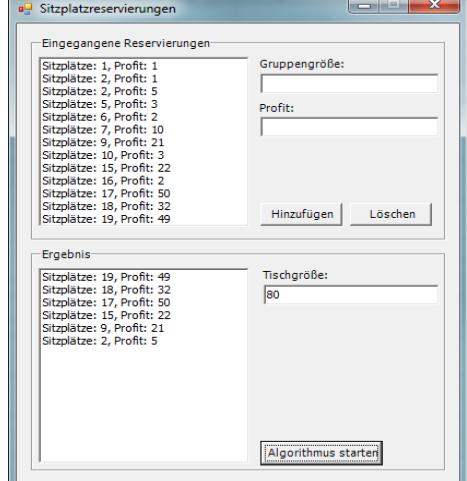
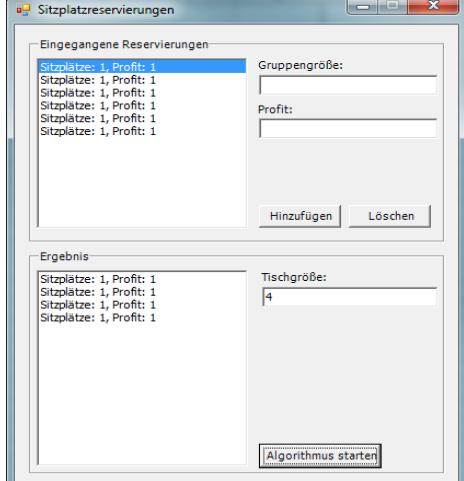
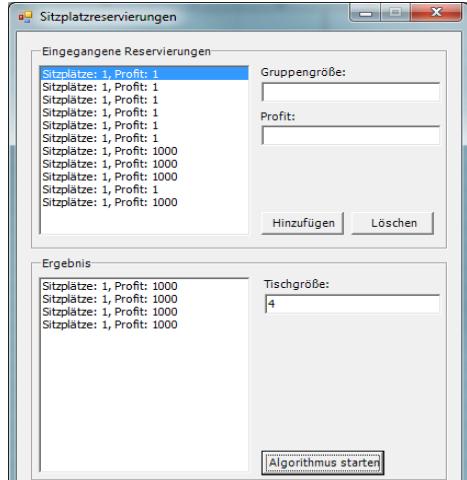
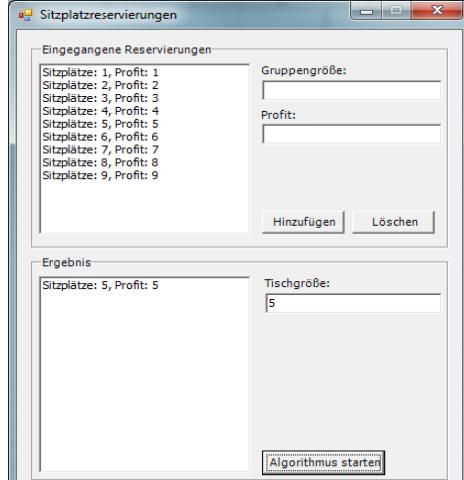
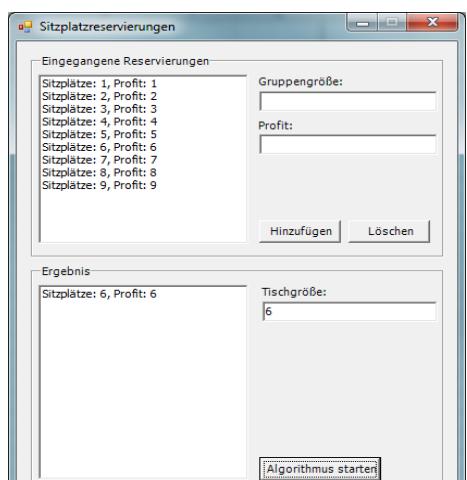
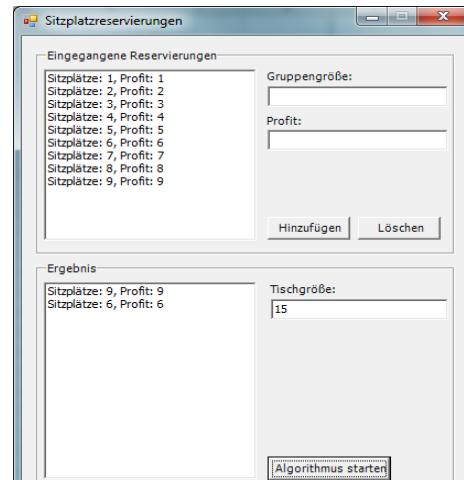
Personen	1	2	2	5	6	7	9	10	15	16	17	18	19	"X" bedeutet, dass eine Gruppe ausgewählt wurde.	
Profit	1	1	5	3	2	10	21	3	22	2	50	32	49		
Tischgröße														Profit	Besetzt
1	X													1	1
2			X											5	2
3	X	X												6	3
4	X	X	X											6	3
5	X	X	X											7	5
6	X	X	X											7	5
7							X							10	7
8	X						X							11	8
9								X						21	9
10	X							X						22	10
11			X				X							26	11
12	X	X					X							27	12
13	X	X					X							27	12
14	X	X	X				X							28	14
15	X	X	X				X							28	14
16						X	X							31	16
17										X				50	17
18	X									X				51	18
19			X							X				55	19
20	X		X							X				56	20
21	X		X							X				56	20
22	X	X	X							X				57	22
23	X	X	X							X				57	22
24						X				X				60	24
25	X					X				X				61	25
26							X			X				71	26
27	X						X			X				72	27
28			X				X			X				76	28
29	X	X					X			X				77	29
30		X	X				X			X				77	30
31	X	X	X				X			X				78	31
32	X	X	X				X			X				78	31
33						X	X			X				81	33
34	X					X	X			X				82	34
35			X			X	X			X				86	35
36										X	X			99	36
37	X									X	X			100	37
38			X							X	X			104	38
39	X	X								X	X			105	39
40		X	X							X	X			105	40
80			X				X	X		X	X	X		179	80
127	X	X	X	X	X	X	X	X	X	X	X	X		201	127

Hier wäre auch diese Gruppe möglich (Profit gleich) statt 1-1.

Gruppe 50-17 ist immer dabei: gutes Profit-Personen-

Keine passende Gruppe mehr, gleiche Gruppen wie zuvor.

### Sitzplatzreservierungen: Einige Beispielefälle

### 3.6 Kontostand für jede Person

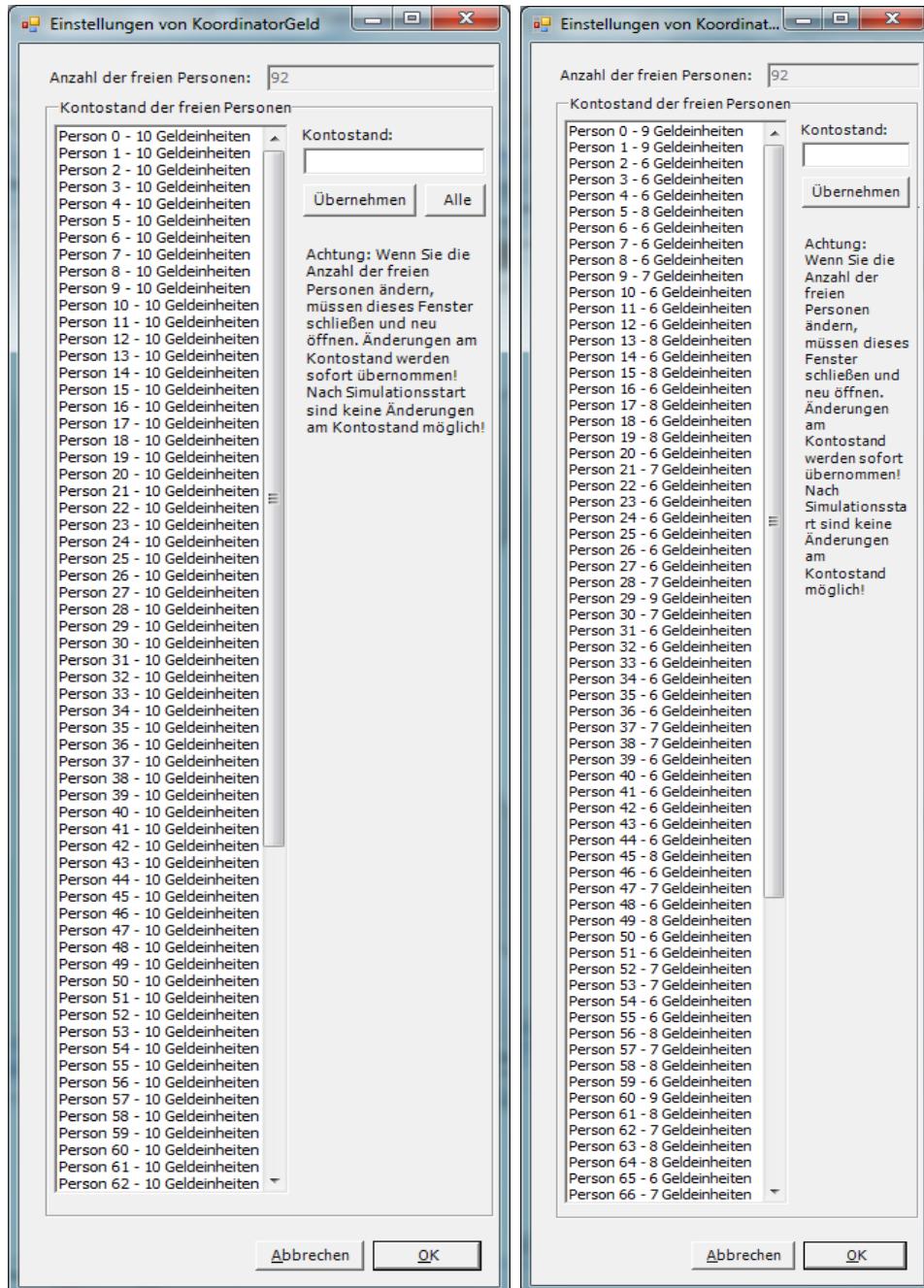
Die folgenden Screenshots zeigen auf der linken Seite den Kontostand der Personen vor Anwendung des Algorithmus und auf der rechten Seite den Kontostand, nachdem der Ober sich geärgert hat.

Ich muss zugeben, dass man sich bei dieser Erweiterung wirklich streiten kann, inwiefern sie sinnvoll ist. Auch hätte man den Algorithmus für die eigentliche Aufgabenstellung so anpassen können, dass die Ergebnisse besser wären. Das schien mir jedoch zu viel Aufwand für eine Erweiterung da ich wohl eine komplett neue Lösungsidee für das *Entfernen der Gruppen* gebraucht hätte<sup>53</sup>.

---

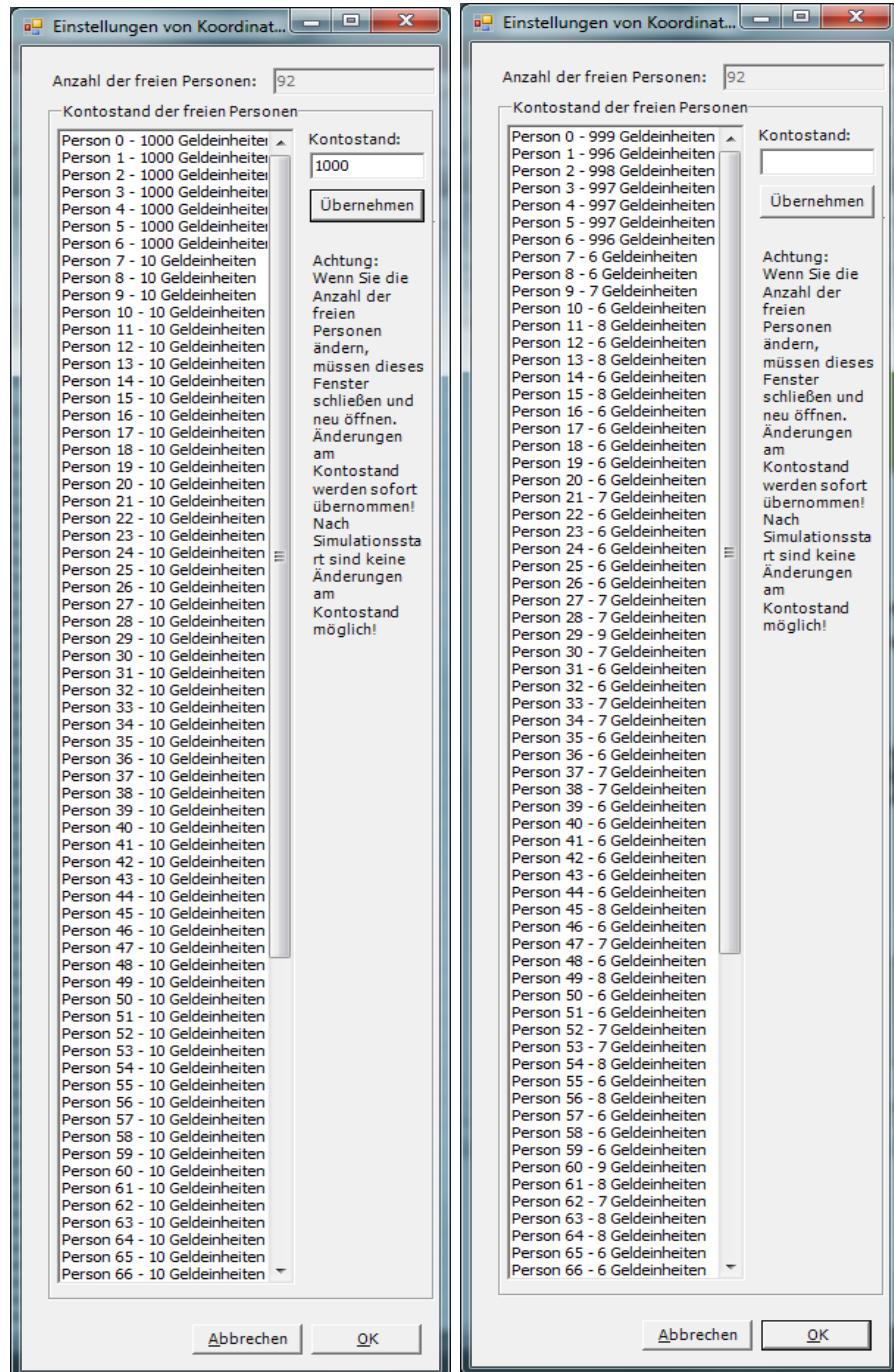
<sup>53</sup>Zumindest ist mir auf die Schnelle keine effiziente Lösung eingefallen.

Koordinator Geld, Anzahl der Personen: 92, Tischgröße: 300



Ganz zu Beginn haben alle Personen den Kontostand 10. Nachdem der Ober sich geärgert hat, haben alle Personen ein Guthaben zwischen 8 und 6, nur zwei Personen haben ein Guthaben von 9. Die erste Gruppe, die sich ganz am Rand befindet, wird zum Beispiel nie entfernt und bleibt bis zum Schluss erhalten. Das lässt sich mit diesem Algorithmus leider nicht vermeiden (gemeint ist die erste Gruppe, die erstellt wird).

Koordinator Geld, Anzahl der Personen: 92, Tischgröße: 300

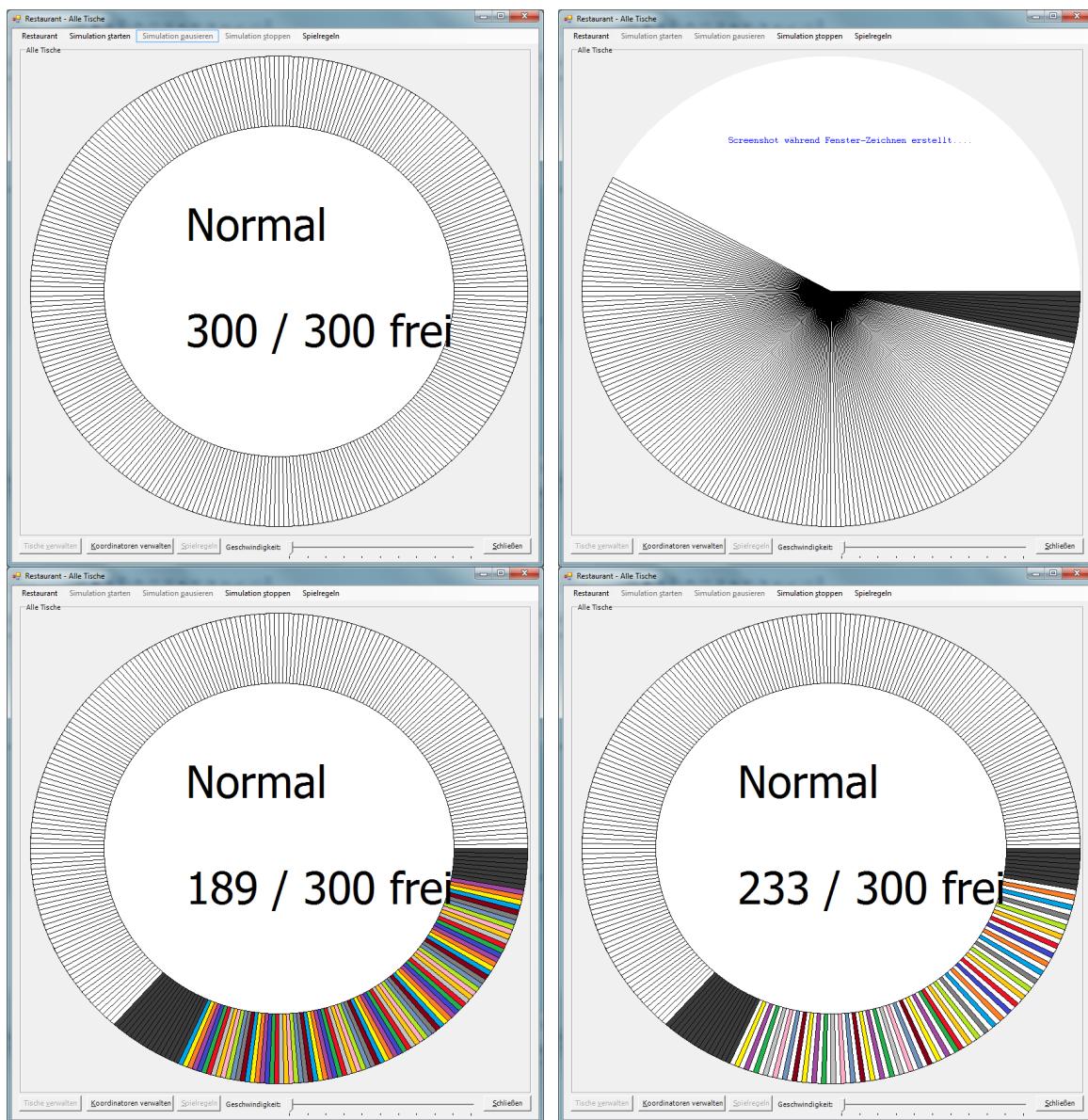


Bei der Person mit dem Kontostand 999 handelt es sich wieder um die erste platzierte Gruppe, die nie entfernt wurde. Ansonsten weisen bis auf eine Person (998) alle Personen mit vorherigem Guthaben 1000 solche Kontostände auf, dass die Person 3-5 Mal platziert wurde. Bei den normalen Personen (Guthaben 10) kommt es hingegen öfter vor, dass eine Gruppe nur zwei Mal verwendet wurde (Kontostand 8).

### 3.7 Normale Personen

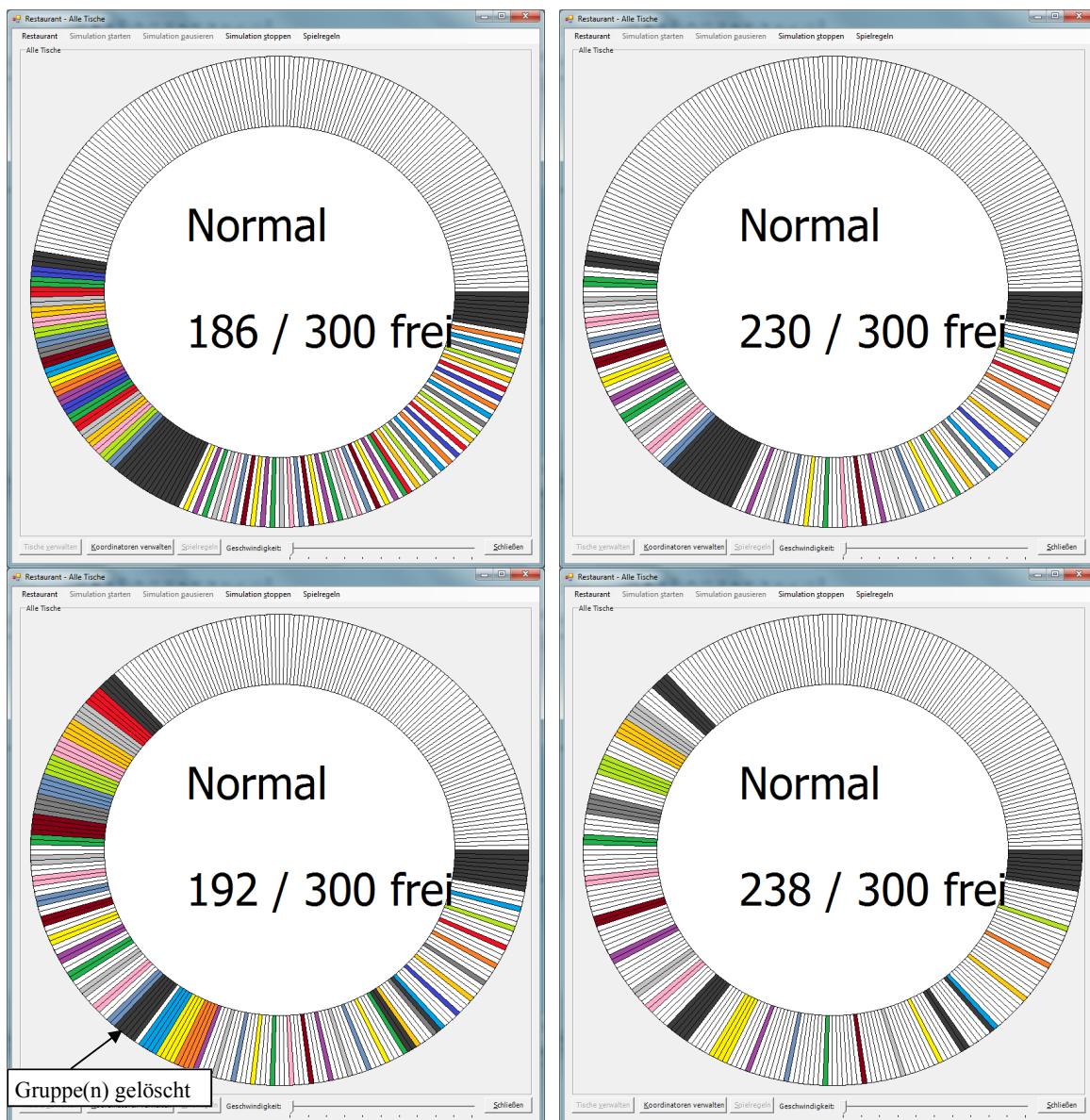
Wenn das Restaurant auch von normalen Personen besucht wird (was in der Regel wohl der Fall sein wird), ist es Zufall, ob der Koordinator gewinnt oder verliert. Dadurch, dass durch das Verschwinden großer Gruppen große Lücken entstehen können, kann es sein, dass der Koordinator verliert. Es kann aber genauso gut sein, dass er nun mit weniger freien Personen gewinnt (im Gegensatz zum Fall ohne die Erweiterung), weil mehr Plätze besetzt sind. Die folgenden zwei Beispiele belegen das. Es ist übrigens auch interessant, wie der Ober die entstehenden Lücken nutzt, indem er kleine normale Gruppen oder Schülergruppen darin platziert. Kleine Gruppen aus normalen Personen (z.B. 2 Personen) können zum Beispiel in den Lücken platziert werden, die der Koordinator erzeugt hat.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30.



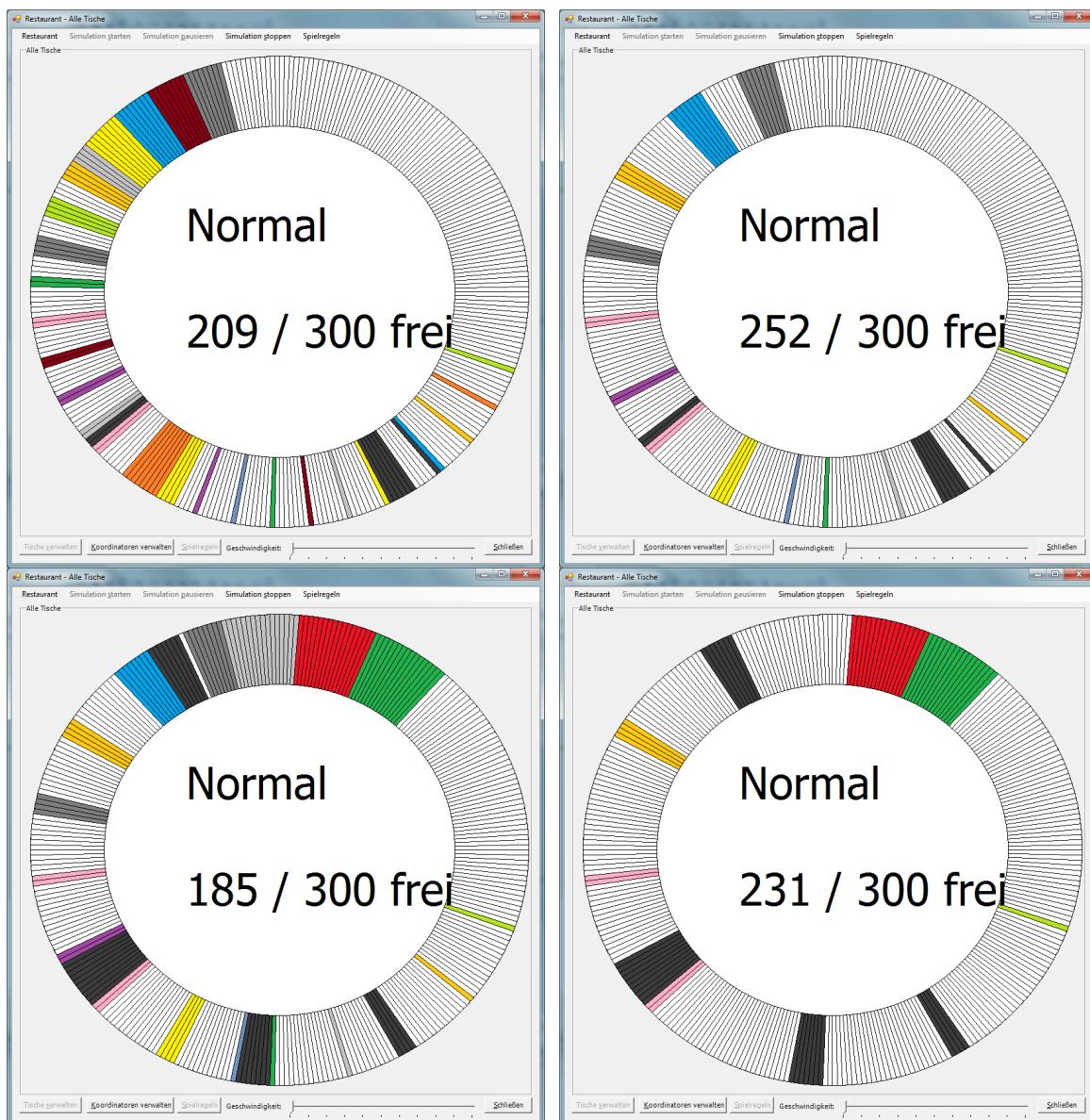
Die Plätze, die von normalen Personen besetzt sind, sind dunkelgrau eingefärbt. Aus dieser Ansicht kann man leider nicht entnehmen, wie die normalen Personen gruppiert sind.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Seite 2.



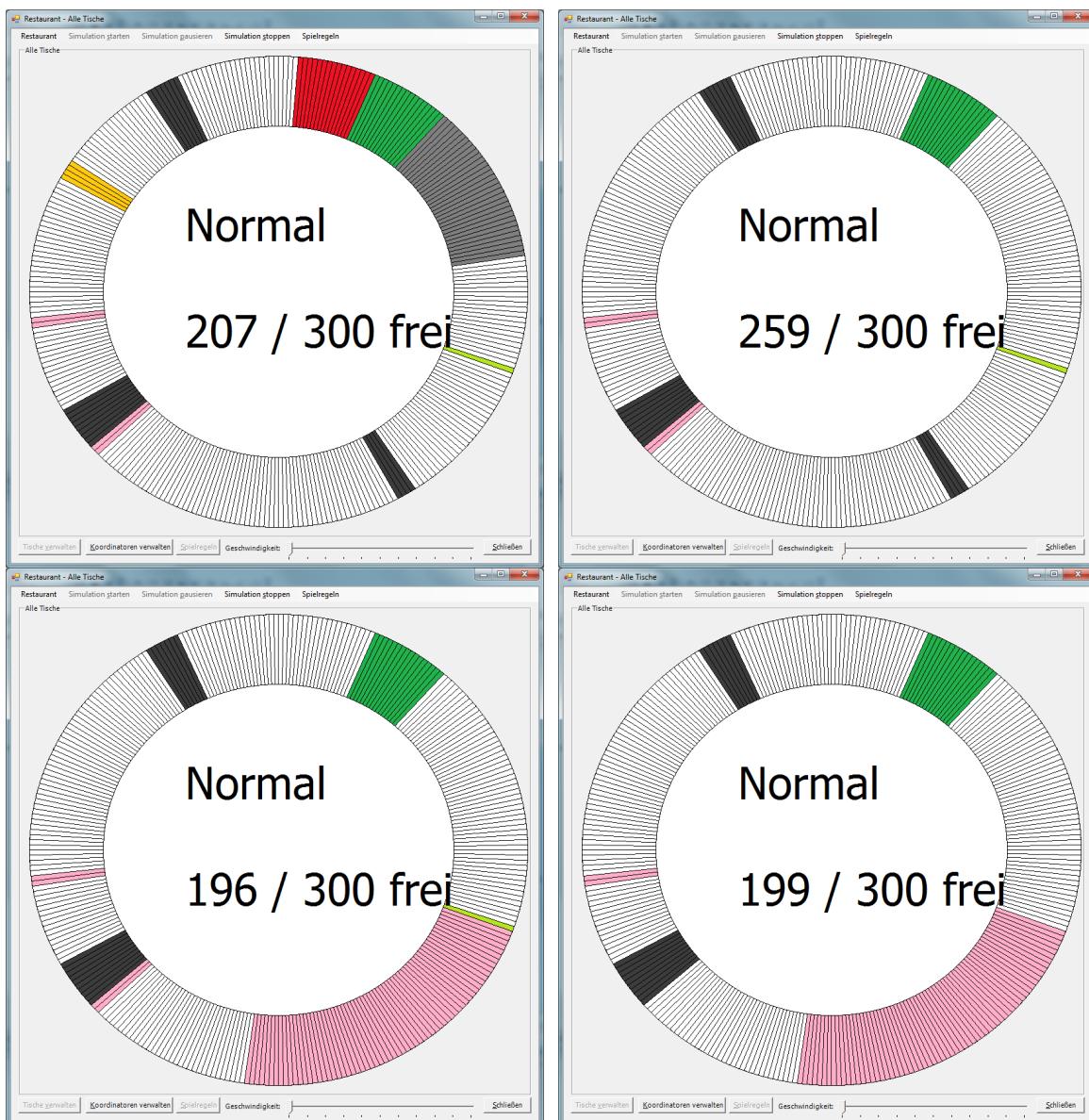
Wenn man genau aufpasst, kann man immer wieder Stellen erkennen, wo Gruppen aus normalen Personen entfernt werden. Es können jederzeit Gruppen hinzugefügt und/oder entfernt werden, egal ob gerade der Ober oder der Koordinator dran ist.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Seite 3.



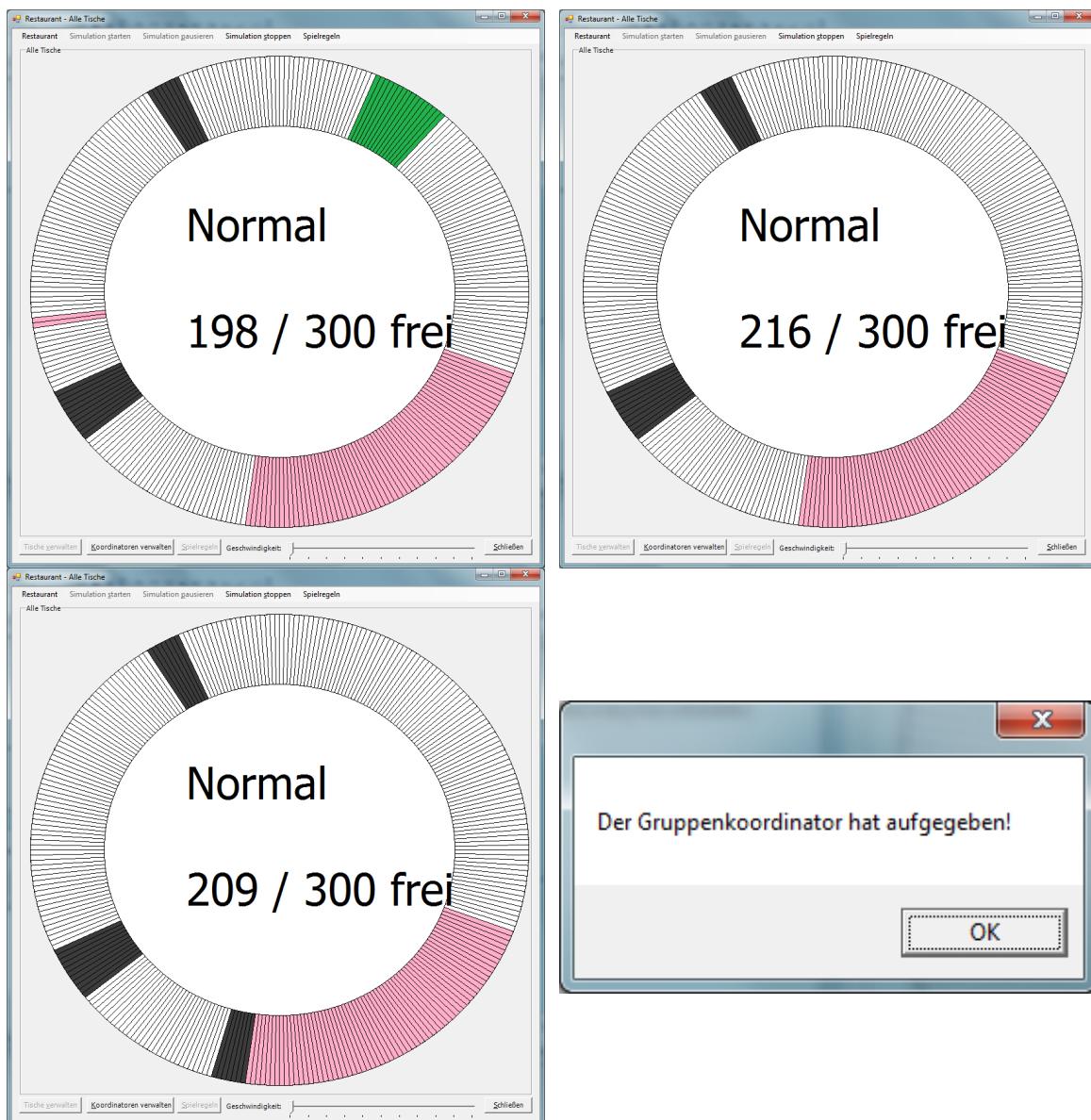
Es werden häufig normale Gruppen gelöscht. Das macht es dem Koordinator schwer, weil zu große Lücken unvorhersehbar entstehen können.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Seite 4.



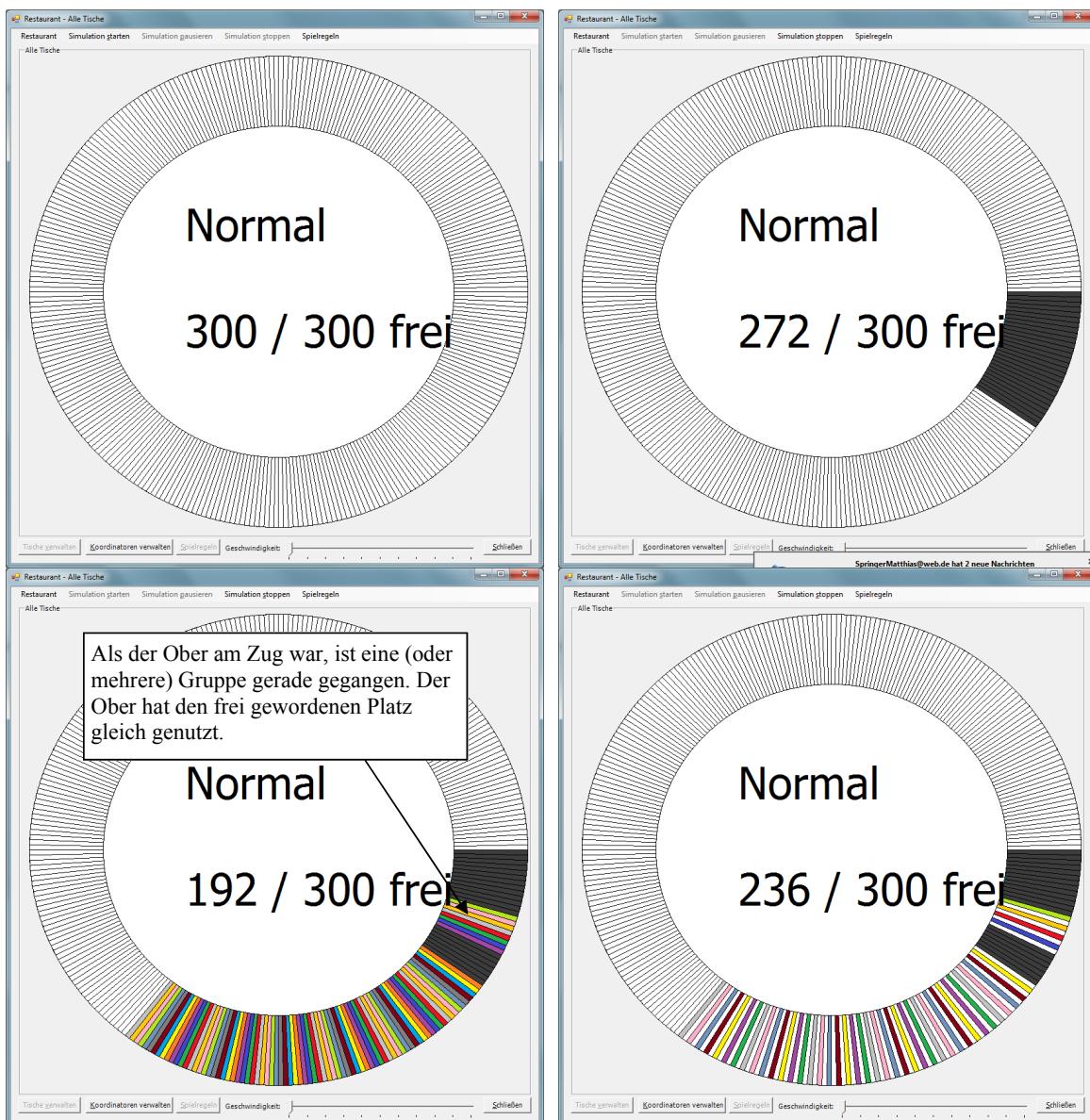
Bei einem so großen Beispiel kann man schnell den Überblick verlieren. Dieses Beispiel soll nur zeigen, wie normale Gruppen zufällig entstehen und verschwinden können, und welche Auswirkungen das auf die anderen Algorithmen hat.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Seite 5.



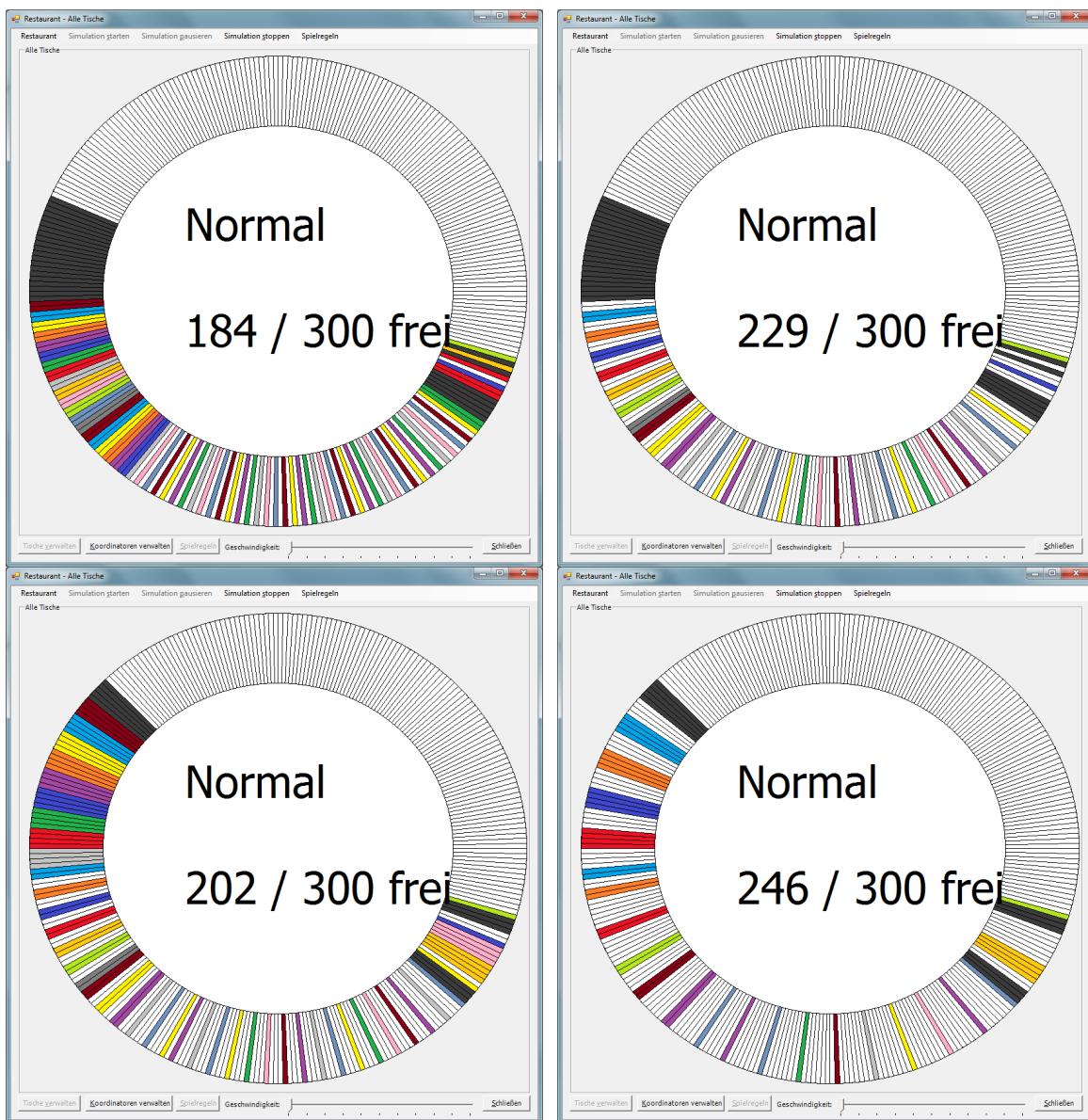
Der Koordinator gibt erst auf, als die Größe der zu erstellenden Gruppen 128 erreicht hat, weil dann die Anzahl aller Personen (Schüler) kleiner als die Größe der zu erstellenden Gruppe ist. Deshalb gibt es so viele Schritte.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Zweiter Versuch.

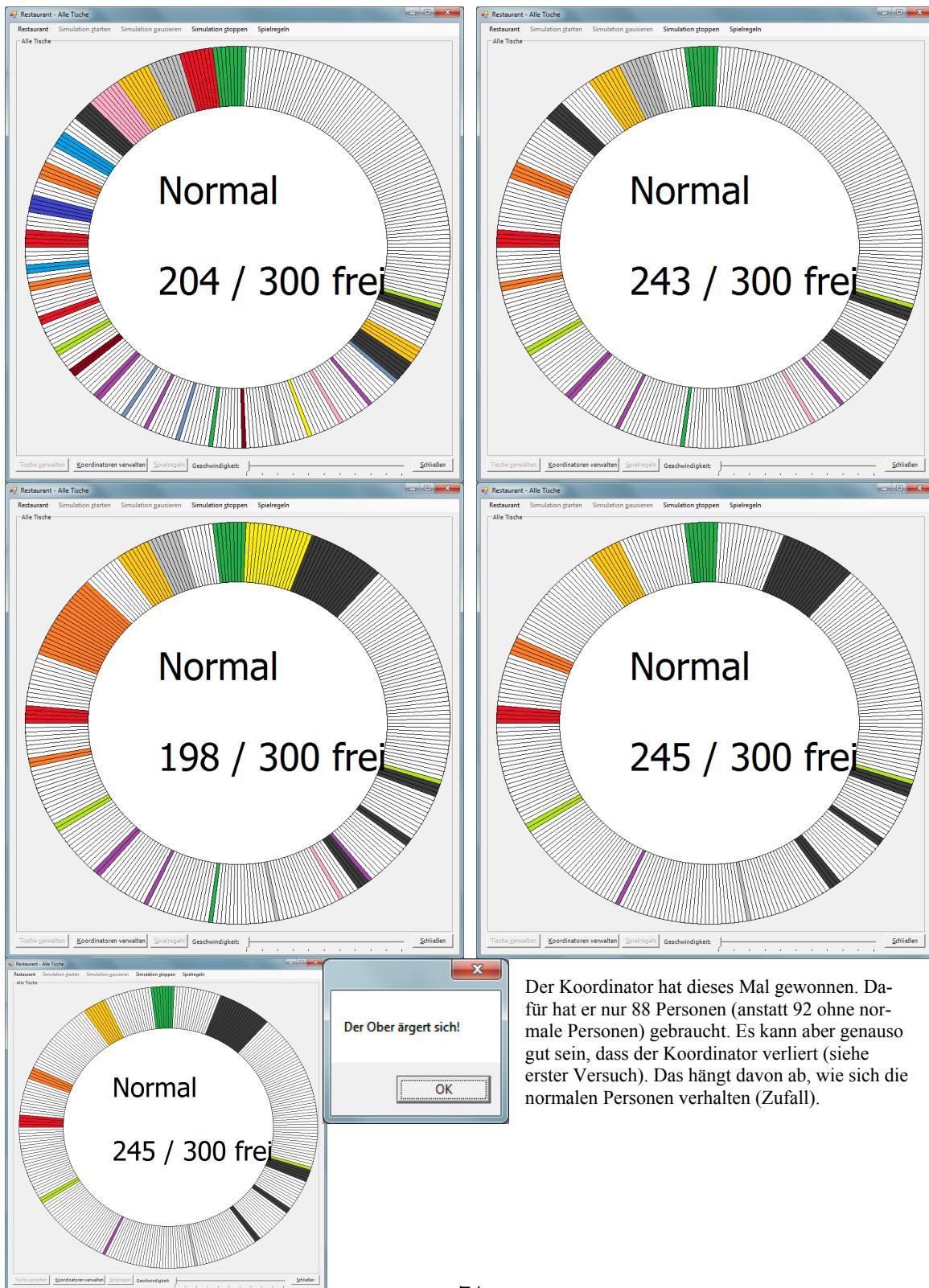


Die OnTick-Aufrufe für den Ober/Koordinator und die Erweiterung „Normale Personen“ erfolgen nicht unbedingt sequentiell. Es kann vorkommen, dass die Erweiterung „Normale Personen“ und der Ober gleichzeitig Aktionen ausführen. Beim Koordinator/Ober wurde das verhindert, indem das Programm wartet, bis der letzte OnTick-Aufruf zu Ende ist.

Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Zweiter Versuch. Seite 2.



Tischgröße: 300, Anzahl der freien Personen: 88, Normale Personen: 30. Zweiter Versuch. Seite 3.



Der Koordinator hat dieses Mal gewonnen. Dafür hat er nur 88 Personen (anstatt 92 ohne normale Personen) gebraucht. Es kann aber genauso gut sein, dass der Koordinator verliert (siehe erster Versuch). Das hängt davon ab, wie sich die normalen Personen verhalten (Zufall).

## 4 Programm-Quelltext

```
F:\Users\meister\Documents\Visual Studio ...\\28_2_bwinf_2\\28_2_bwinf_2\\Program.cs 1
1 using System;
2 using System.Collections.Generic;
3 using System.Windows.Forms;
4 using _28_2_bwinf_2.GUI;
5 using _28_2_bwinf_2.RestaurantAlgorithmen;
6 using _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen;
7
8 namespace _28_2_bwinf_2
9 {
10    class Program
11    {
12        public static TischeVerwalten TischeVerwaltenGUI = new TischeVerwalten();
13        public static RestaurantGUI RestaurantTische;
14        public static Restaurant GlobalRestaurant;
15        public static ChefsVerwalten ChefsVerwaltenGUI = new ChefsVerwalten();
16
17        [STAThread]
18        static void Main(string[] args)
19        {
20            RestaurantTische = new RestaurantGUI();
21
22            ErstelleNeuesRestaurant();
23            GlobalRestaurant.ChefOber = new EinfacherChefOber(GlobalRestaurant);
24            GlobalRestaurant.GruppenKoordinator = new KoordinatorGeld(GlobalRestaurant);
25
26            RestaurantTische.DasRestaurant = GlobalRestaurant;
27
28            Application.Run(RestaurantTische);
29        }
30
31        public static void ErstelleNeuesRestaurant()
32        {
33            GlobalRestaurant = new Restaurant();
34        }
35    }
36}
37
```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\\Interfaces.cs 1
1 #region Assemblies
2
3 using System;
4 using System.Collections.Generic;
5 using System.Drawing;
6 using System.IO;
7 using System.Timers;
8 using _28_2_bwinf_2.GUI;
9 using _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen;
10 using Timer=System.Timers.Timer;
11
12 #endregion
13
14 namespace _28_2_bwinf_2.RestaurantAlgorithmen
15 {
16     /// <summary>
17     /// Enthaelt statische Objekte, die alle Klassen immer verwenden koennen.
18     /// </summary>
19     public static class StatischeObjekte
20     {
21         // Diese Funktionen sind nur dazu da, um mir beim Fehler suchen zu helfen
22         public static void ExceptionWerfenGUIEreignisListeGefuellt(object dasObjekt)
23         {
24             throw new Exception("Die GUI-Ereignis-Liste fuer das aktuelle Objekt
enthaelt noch Ereignisse (ist nicht leer).");
25         }
26
27         public static void DebugNachricht(string text)
28         {
29             StreamWriter datei = new StreamWriter("log.txt", true);
30
31             try
32             {
33                 Console.WriteLine("[" + Program.GlobalRestaurant.Uhrzeit.ToString() + ↵
" ] " + text);
34                 datei.WriteLine("[" + Program.GlobalRestaurant.Uhrzeit.ToString() + "] " ↵
" " + text);
35             }
36             catch
37             {
38                 // Es existiert kein GlobalRestaurant-Objekt
39                 Console.WriteLine("[?] " + text);
40                 datei.WriteLine("[?] " + text);
41             }
42
43             datei.Close();
44         }
45     }
46
47     #region Interfaces
48     /// <summary>
49     /// Alle Personen im Restaurant (Ober, Gruppen, usw.).
50     /// </summary>
51     public interface IPersonImRestaurant
52     {
53         Restaurant DasRestaurant
54         {
55             get;
56             set;
57         }
58
59         void SimulationStopp();
60         void SimulationPause();
61         void SimulationStart();
62
63         void GuiEreignisseCallBackZeigerLoeschen();
64         void GuiEreignisseSetzen();
65
66         SteuerelementEinstellungen GuiEinstellungen();
67     }

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 2

68
69  /// <summary>
70  /// Ein Ober ist fuer einen einzelnen Tisch zustaendig. Sein Vorgesetzter ist der Chefober. ↵
71  /// </summary>
72  public interface IOber : IPersonImRestaurant
73  {
74      GruppePositionAmTisch GruppeKommt(IGruppe gruppe);
75      void GruppeGeht(IGruppe gruppe);
76      void AnweisungAusfuehren(OberAktion aktion, List<int> parameter, IGruppe gruppe, ↵
77      , GruppePositionAmTisch positionAmTisch);
78
79      Tisch EigenerTisch
80      {
81          get;
82          set;
83      }
84
85  /// <summary>
86  /// Der Chefober koordiniert alle Vorgaenge im Restaurant, wenn mehrere Tische vorhanden sind. ↵
87  /// </summary>
88  public interface IChefOber : IPersonImRestaurant
89  {
90      GruppePositionAmTisch GruppeKommt(IGruppe gruppe);
91      void GruppeGeht(IGruppe gruppe);
92      bool ErlaubnisEinholen(OberAktion aktion, List<int> parameter, IGruppe gruppe, ↵
93      GruppePositionAmTisch positionAmTisch);
94
95  /// <summary>
96  /// Eine Gruppe besteht aus mehreren Personen und will einen zusammenhaengenden Platzbereich. ↵
97  /// </summary>
98  public interface IGruppe : IPersonImRestaurant, IComparable
99  {
100     /// <summary>
101     /// Gibt an, wie lange sich eine Gruppe bereits im Restaurant befindet.
102     /// </summary>
103     int BisherigeAufenthaltsdauer();
104
105     /// <summary>
106     /// Gibt an, wie lange die Gruppe noch auf Sitzplaetze wartet, bis sie nach Hause geht. ↵
107     /// </summary>
108     int WartezeitSitzplatz();
109
110     /// <summary>
111     /// Loest die Gruppe auf. Entfernt alle auf das Objekt zeigende Ereignisse.
112     /// </summary>
113     void GruppeAufloesen();
114
115     /// <summary>
116     /// Gibt "wahr" zurueck, wenn es sich um eine Gruppe handelt, die aus Schuelern bestehende und von einem Koordinator verwaltet wird.
117     /// </summary>
118     bool IstSchuelerGruppe();
119
120     bool IstGruppeSchonGegangen();
121
122     int EindeutigerIndex();
123
124     GruppePositionAmTisch PositionAmTisch
125     {
126         get;
127         set;
128     }
129
130     bool IstGruppePlatziert

```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Interfaces.cs 3
131     {
132         get;
133         set;
134     }
135
136     List<IPerson> Personen
137     {
138         get;
139         set;
140     }
141
142     System.Drawing.Color GuiGruppenFarbe
143     {
144         get;
145         set;
146     }
147 }
148
149 public interface IPerson : IComparable
150 {
151     PersonPositionAmTisch PositionAmTisch
152     {
153         get;
154         set;
155     }
156
157     string Name
158     {
159         get;
160         set;
161     }
162 }
163
164 /// <summary>
165 /// Der Gruppenkoordinator bildet Gruppen aus einzelnen Personen und gibt diesen Anweisungen.
166 /// </summary>
167 public interface IGruppenKoordinator : IPersonImRestaurant
168 {
169     bool ErlaubnisEinholen(GruppeAktion aktion, IGruppe gruppe, List<int> parameter);
170
171     int AnzahlDerPersonen
172     {
173         get;
174         set;
175     }
176 }
177 #endregion
178
179 #region Strukturen (Wertetypen)
180 /// <summary>
181 /// Die Position einer Gruppe im Restaurant.
182 /// </summary>
183 [Serializable]
184 public struct GruppePositionAmTisch
185 {
186     public string GruppenName;
187     public int GruppePositionStart;
188     public int GruppePositionEnde;
189     public int GruppeGroesse;
190     public Tisch GruppeTisch;    // Achtung: Referenztyp!
191 }
192
193 /// <summary>
194 /// Die Position einer Person im Restaurant
195 /// </summary>
196 [Serializable]
197 public struct PersonPositionAmTisch
198 {

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 4
199     public int Tischnummer;
200     public int Position;
201 }
202 #endregion
203
204 #region Aufzaehlungen
205 /// <summary>
206 /// Aktionen, die ein Ober ausfuehren kann.
207 /// </summary>
208 public enum OberAktion
209 {
210     Nichts, PersonPlatzieren, PersonUmsetzen, PersonHinauswerfen, GruppePlatzieren
211 }
212
213 /// <summary>
214 /// Aktionen, die eine Gruppe ausfuehren kann.
215 /// </summary>
216 public enum GruppeAktion
217 {
218     Nichts, TischVerlassen
219 }
220 #endregion
221
222 #region Funktionszeiger fuer Ereignisse
223 public delegate void OberAergertSichEreignis(IOber ober, IChefOber chefOber,
224 IGruppe gruppe, List<int> parameter);
225 public delegate void GruppePlatziertEreignis(IOber ober, IGruppe gruppe);
226 public delegate void TickEreignis(long zeit);
227 public delegate void GuiTischNeuZeichnenEreignis();
228
229 public delegate void KoordinatorAufgebenEreignis(IGruppenKoordinator koordinator);
230 #endregion
231
232 #region Klassen (Referenztypen)
233 /// <summary>
234 /// Das Restaurant mit Chefober, Tischen und einem Gruppenkoordinator. Ueber dieses
235 /// Objekt koennen alle beteiligten Personen Informationen abfragen.
236 /// </summary>
237 [Serializable]
238 public class Restaurant
239 {
240     private object _chefOber;
241     private object _gruppenKoordinator;
242     public List<Tisch> ListeDerTische;
243     public int Intervall = 10000;
244     public int Uhrzeit = 0;
245     public NormalePersonen normalePersonen;
246
247     private bool _letzterTimerAufrufSchonFertig = true;
248
249     [NonSerialized]
250     private Timer _tickTimer;
251
252     public IChefOber ChefOber
253     {
254         get
255         {
256             return (IChefOber)_chefOber;
257         }
258         set
259         {
260             _chefOber = value;
261         }
262     }
263
264     public IGruppenKoordinator GruppenKoordinator
265     {
266

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs      5
267         get
268     {
269         return (IGruppenKoordinator)_gruppenKoordinator;
270     }
271     set
272     {
273         _gruppenKoordinator = value;
274     }
275 }
276
277 #region Ereignisse im Restaurant
278 public event GruppePlatziertEreignis GruppePlatziert;
279 public event OberAergertSichEreignis OberAergertSich;
280 public event KoordinatorAufgebenEreignis KoordinatorAufgeben;
281
282 [field: NonSerialized]
283 public event OberAergertSichEreignis GUI_OberAergertSich;
284 [field: NonSerialized]
285 public event KoordinatorAufgebenEreignis GUI_KoordinatorAufgeben;
286
287 /// <summary>
288 /// Dieses Ereignis wird ausgelöst, wenn eine Zeiteinheit vergangen ist.
289 /// </summary>
290 public event TickEreignis Tick;
291 public event TickEreignis GUI_Tick;
292
293 /// <summary>
294 /// Muss nach der Deserialisierung aufgerufen werden, da das Timer-Objekt nicht
295 serialisiert werden kann.
296 /// </summary>
297 public void NachDeserialisierung()
298 {
299     _tickTimer = new Timer();
300     _tickTimer.Interval = Intervall;
301 }
302
303     public void EreignisAusloesenOberAergertSich(IOber ober, IChefOber chefOber,
304 IGruppe gruppe, List<int> parameter)
305     {
306         try
307         {
308             OberAergertSich(ober, chefOber, gruppe, parameter);
309         }
310     }
311
312     GUI_OberAergertSich(ober, chefOber, gruppe, parameter);
313 }
314
315     public void EreignisAusloesenGruppePlatziert(IOber ober, IGruppe gruppe)
316     {
317         GruppePlatziert(ober, gruppe);
318     }
319
320     public void EreignisAusloesenKoordinatorAufgeben(IGruppenKoordinator
321 koordinator)
322     {
323         try
324         {
325             KoordinatorAufgeben(koordinator);
326         }
327         catch (NullReferenceException)
328         {
329
330         }
331
332         GUI_KoordinatorAufgeben(koordinator);
333     }

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 6

334     public void GuiEreignisseCallBackZeigerLoeschen()
335     {
336         // Nicht mehr notwendig!
337     }
338
339
340     public void GuiEreignisseSetzen()
341     {
342         // Nicht mehr notwendig!
343     }
344
345     /// <summary>
346     /// Tick-Funktion fuer den Timer, die die Uhrzeit erhoeht und das oeffentliche Ereignis ausloest.
347     /// </summary>
348     private void TimerTick(object sender, ElapsedEventArgs e)
349     {
350         if (_letzterTimerAufrufSchonFertig)
351         {
352             _letzterTimerAufrufSchonFertig = false;
353             Uhrzeit++;
354             Tick(Uhrzeit);
355             _letzterTimerAufrufSchonFertig = true;
356         }
357     }
358 #endregion
359
360     /// <summary>
361     /// Startet die Simulation und startet den Timer.
362     /// </summary>
363     public void SimulationStart()
364     {
365         // Timer initialisieren
366         _tickTimer = new Timer { Interval = Intervall, AutoReset = true };
367         _tickTimer.Enabled = true;
368         ((IPersonImRestaurant)_gruppenKoordinator).SimulationStart();
369         ((IPersonImRestaurant)_chefOber).SimulationStart();
370         _tickTimer.Elapsed += new ElapsedEventHandler(TimerTick);
371         _tickTimer.Enabled = true;
372     }
373
374     public void SimulationPause()
375     {
376         _tickTimer.Enabled = false;
377         ((IPersonImRestaurant)_gruppenKoordinator).SimulationPause();
378         ((IPersonImRestaurant)_chefOber).SimulationPause();
379     }
380
381     public void SimulationStopp()
382     {
383         _tickTimer.Enabled = false;
384         ((IPersonImRestaurant)_gruppenKoordinator).SimulationStopp();
385         ((IPersonImRestaurant)_chefOber).SimulationStopp();
386     }
387
388     /// <summary>
389     /// Konstruktor der Restaurant-Klasse, der alle Listen intialisiert.
390     /// </summary>
391     public Restaurant()
392     {
393         ListeDerTische = new List<Tisch>();
394         normalePersonen = new NormalePersonen(this);
395     }
396
397
398     /// <summary>
399     /// Eine Person.
400     /// </summary>
401     [Serializable]
402     public class Person : IPerson

```

```
F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Interfaces.cs 7
403     {
404         private PersonPositionAmTisch _positionAmTisch;
405         private string _name;
406
407         public Person()
408         {
409
410         }
411
412         public Person(string name)
413         {
414             Name = name;
415         }
416
417         public override bool Equals(object obj)
418         {
419             Person p = obj as Person;
420
421             if ((object)p == null)
422             {
423                 return false;
424             }
425
426             return p.Name == Name;
427         }
428
429         public virtual int CompareTo(object obj)
430         {
431             // Dummy
432             throw new NotImplementedException();
433         }
434
435         public static bool operator ==(Person a, Person b)
436         {
437             return a.Equals(b);
438         }
439
440         public static bool operator !=(Person a, Person b)
441         {
442             return !(a == b);
443         }
444
445         public PersonPositionAmTisch PositionAmTisch
446         {
447             get
448             {
449                 return _positionAmTisch;
450             }
451             set
452             {
453                 _positionAmTisch = value;
454             }
455         }
456
457         public string Name
458         {
459             get
460             {
461                 return _name;
462             }
463             set
464             {
465                 _name = value;
466             }
467         }
468     }
469
470     /// <summary>
471     /// Ein Tisch mit einem Ober.
472     /// </summary>
```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 8
473     [Serializable]
474     public class Tisch : IComparable
475     {
476         /// <summary>
477         /// Speichert die Groesse des Tisches als numerischen Wert.
478         /// </summary>
479         private int _tischgroesse = 0;
480
481         /// <summary>
482         /// Speichert einen Zeiger auf das Restaurant.
483         /// </summary>
484         private Restaurant _meinRestaurant;
485
486         /// <summary>
487         /// Legt fest, welchen Namen dieser Tisch tragen soll. Wird nur von der GUI fuer optische Zwecke verwendet.
488         /// </summary>
489         private string _guiTischBezeichner = "Unbenannter Tisch";
490
491         /// <summary>
492         /// Wenn dieser Wert "wahr" ist, zeichnet die GUI diesen Tisch bei jedem Tick, egal, ob sich etwas veraendert hat, oder nicht.
493         /// </summary>
494         private bool _guiZeichneGuiBeiJedemTick = true;
495
496         public int Tischnummer;
497
498         /// <summary>
499         /// Speichert eine Liste aller Gruppen.
500         /// </summary>
501         private List<IGruppe> _gruppen;
502
503         /// <summary>
504         /// Enthaelt den fuer diesen Tisch zustaendigen Ober.
505         /// </summary>
506         private IOber _ober;
507
508         /// <summary>
509         /// Speichert ein Array, das jedem Sitzplatz eine Gruppe zuweist.
510         /// </summary>
511         private IGruppe[] _sitzplaetze;
512
513         /// <summary>
514         /// Das Tick-Ereignis wird aufgerufen, wenn eine Zeiteinheit vergangen ist.
515         /// </summary>
516         private TickEreignis _tickEreignis;
517
518         /// <summary>
519         /// Legt fest, ob ein Platz besetzt ist.
520         /// </summary>
521         private bool[] _istPlatzBesetzt;
522
523         [field:NonSerialized] public event GuiTischNeuZeichnenEreignis GUI_NeuZeichnen;
524
525         public IOber Ober
526         {
527             get
528             {
529                 return (IOber)_ober;
530             }
531             set
532             {
533                 _ober = value;
534             }
535         }
536
537         public List<IGruppe> Gruppen
538         {
539             get
540             {

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 9
541         return _gruppen;
542     }
543     set
544     {
545         _gruppen = value;
546     }
547 }
548
549     public IGruppe GetSitzplaetze(int index)
550     {
551         if (!_istPlatzBesetzt[index])
552         {
553             return default(IGruppe);
554         }
555         else
556         {
557             return _sitzplaetze[index];
558         }
559     }
560
561     public void SetSitzplaetze(int index, IGruppe gruppe)
562     {
563         if (gruppe == default(IGruppe))
564         {
565             _istPlatzBesetzt[index] = false;
566             _sitzplaetze[index] = default(IGruppe);
567         }
568         else
569         {
570             _istPlatzBesetzt[index] = true;
571             _sitzplaetze[index] = gruppe;
572         }
573     }
574
575     /// <summary>
576     /// Achtung: Diesen Konstruktur nur fuer Serialisierung verwenden!
577     /// </summary>
578     public Tisch()
579     {
580
581     }
582
583     /// <summary>
584     /// Der Destruktor entfernt den verwaisten Funktionszeiger auf die OnTick-
585     /// Funktion.
586     /// </summary>
587     ~Tisch()
588     {
589         EntferneTickEreignis();
590     }
591
592     /// <summary>
593     /// Vergleicht zwei Tische. Ein Tisch ist "groesser", wenn er mehr freie
594     /// Plaetze hat.
595     /// </summary>
596     public int CompareTo(object obj)
597     {
598         Tisch tisch = (Tisch) obj;
599         return FreiePlaetze() - tisch.FreiePlaetze();
600     }
601
602     /// <summary>
603     /// Der Tischbezeichner wird nur fuer die GUI benoetigt, um die Tische leichter
604     /// unterscheiden zu koennen.
605     /// </summary>
606     public string GuiTischBezeichner
607     {
608         get
609         {
610             return _guiTischBezeichner;
611         }
612     }

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 10
608         }
609
610         set
611     {
612         _guiTischBezeichner = value;
613     }
614 }
615
616 /// <summary>
617 /// Weist die GUI an, den Tisch bei jedem Tick neu zu zeichnen. Nicht zu ↵
617 empfehlen bei sehr kleinen Tick-Intervallen, weil dann sehr viel Zeit verloren geht ↵
618     /// </summary>
619     public bool GuiZeichneGuiBeiJedemTick
620     {
621         get
622         {
623             return _guiZeichneGuiBeiJedemTick;
624         }
625
626         set
627         {
628             _guiZeichneGuiBeiJedemTick = value;
629         }
630     }
631
632     public int Tischgroesse
633     {
634         get
635         {
636             return _tischgroesse;
637         }
638
639         set
640         {
641             _tischgroesse = value;
642             _sitzplaetze = new IGruppe[_tischgroesse];
643             _istPlatzBesetzt = new bool[_tischgroesse];
644
645             // Sitzplaetze als nicht besetzt markieren
646             for (int i = 0; i < _tischgroesse; i++)
647             {
648                 _sitzplaetze[i] = default(IGruppe);
649                 _istPlatzBesetzt[i] = false;
650             }
651         }
652     }
653
654     public Restaurant DasRestaurant
655     {
656         get
657         {
658             return _meinRestaurant;
659         }
660
661         set
662         {
663             EntferneTickEreignis();
664             _meinRestaurant = value;
665             _tickEreignis = new TickEreignis(OnTick);
666             _meinRestaurant.Tick += _tickEreignis;
667         }
668     }
669
670     /// <summary>
671     /// Versucht, das Tick-Ereignis bei der Restaurant-Klasse "abzumelden".
672     /// </summary>
673     private void EntferneTickEreignis()
674     {
675         try

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 11
676         {
677             _meinRestaurant.Tick -= _tickEreignis;
678         }
679     catch
680     {
681     }
682 }
683 /// <summary>
684 /// Gibt "wahr" zurueck, wenn der betreffende Sitzplatz noch frei ist.
685 /// </summary>
686 /// <param name="platznummer">Index des Platzes</param>
687 public bool IstPlatzFrei(int platznummer)
688 {
689     return !_istPlatzBesetzt[platznummer];
690 }
691
692
693 /// <summary>
694 /// Ein Sitzplatz wird als "frei" gekennzeichnet. Achtung: Es werden keine
695 weiteren Aenderungen (Aufloesung der Gruppe etc.) vorgenommen. ↵
696 /// </summary>
697 /// <param name="platznummer">Index des Sitzplatzes</param>
698 public void MachePlatzFrei(int platznummer)
699 {
700     SetSitzplaetze(platznummer, default(IGruppe));
701 }
702
703 /// <summary>
704 /// Das Tick-Ereignis wird dann aufgerufen, wenn eine Zeiteinheit vergangen ist ↵
705 /// <param name="uhrzeit">Die aktuelle Uhrzeit</param>
706 private void OnTick(long uhrzeit)
707 {
708     try
709     {
710         if (_guiZeichneGuiBeiJedemTick)
711         {
712             GUI_NeuZeichnen();
713         }
714     }
715     catch
716     {
717     }
718 }
719 }
720
721 public void EreignisAusloesenGUINeuZeichnen()
722 {
723     try
724     {
725         GUI_NeuZeichnen();
726     }
727     catch
728     {
729     }
730 }
731 }
732
733 /// <summary>
734 /// Gibt die Anzahl der freien Sitzplaetze zurueck.
735 /// </summary>
736 public int FreiePlaetze()
737 {
738     int zaehler = 0;
739
740     for (int i = 0; i < _istPlatzBesetzt.Length; i++)
741     {
742         if (!_istPlatzBesetzt[i])
743     }

```

```

F:\Users\meister\Documents\Visual Studio ...RestaurantAlgorithmen\Interfaces.cs 12
744         zaehler++;
745     }
746 }
747
748     return zaehler;
749 }
750
751 /// <summary>
752 /// Zur Sicherheit MUSS ein Zeiger auf das Restaurant uebermittelt werden.
753 /// </summary>
754 /// <param name="restaurant">Zeiger auf das Restaurant</param>
755 public Tisch(Restaurant restaurant)
756 {
757     _meinRestaurant = restaurant;
758     _tickEreignis = new TickEreignis(OnTick);
759     _meinRestaurant.Tick += _tickEreignis;
760 }
761
762 /// <summary>
763 /// Bei einer Umwandlung in eine Zeichenfolge werden Gruppenbezeichner und
764 /// Anzahl der freien Plaetze zurueckgegeben.
765 /// </summary>
766 /// <returns>Gruppenbezeichner (Freie Plaetze / Plaetze insgesamt)</returns>
767 public override string ToString()
768 {
769     return _guiTischBezeichner + " (" + FreiePlaetze().ToString() + " / " +
770     _tischgroesse.ToString() + " frei)";
771 }
772
773 public void GuiEreignisseCallBackZeigerLoeschen()
774 {
775     // Nicht mehr notwendig
776     /*
777     if (_guiEreignisse == default(List<object>)) _guiEreignisse = new List
778     <object>();
779     if (_guiEreignisse.Count != 0) StatischeObjekte.
780     ExceptionWerfenGUIEreignisListeGefuellt(this);
781
782     // Mache das fuer alle Gruppen
783     foreach (IGruppe gruppe in _sitzplaetze)
784     {
785         if (gruppe != default(IGruppe)) gruppe.
786         GuiEreignisseCallBackZeigerLoeschen();
787     }
788
789     _guiEreignisse = new List<object>();
790     _guiEreignisse.Add(GUI_NeuZeichnen);
791
792     GUI_NeuZeichnen = default(GuiTischNeuZeichnenEreignis);
793     */
794 }
795
796 public void GuiEreignisseSetzen()
797 {
798     // Nicht mehr notwendig
799     /*
800     // Mache das fuer alle Gruppen
801     foreach (IGruppe gruppe in _sitzplaetze)
802     {
803         if (default(IGruppe) != gruppe) gruppe.GuiEreignisseSetzen();
804     }
805
806     #endregion
807 }
808

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 1
1 using System;
2 using System.Collections.Generic;
3 using System.Windows.Forms;
4 using _28_2_bwinf_2.GUI;
5
6 namespace _28_2_bwinf_2.RestaurantAlgorithmen
7 {
8     /// <summary>
9     /// Der einfache Ober versucht, Gruppen möglichst effizient zu platzieren, indem möglichst wenig Platz verschwendet wird.
10    /// </summary>
11    [Serializable]
12    public class EinfacherOber : IOber
13    {
14        protected Tisch _meinTisch;
15        protected Restaurant _meinRestaurant;
16
17        public virtual Tisch EigenerTisch
18        {
19            get
20            {
21                return _meinTisch;
22            }
23
24            set
25            {
26                _meinTisch = value;
27            }
28        }
29
30        public virtual Restaurant DasRestaurant
31        {
32            get
33            {
34                return _meinRestaurant;
35            }
36
37            set
38            {
39                _meinRestaurant = value;
40            }
41        }
42
43        public virtual void AnweisungAusfuehren(OberAktion aktion, List<int> parameter, IGruppe gruppe, GruppePositionAmTisch positionAmTisch)
44        {
45            switch (aktion)
46            {
47                case OberAktion.GruppePlatzieren:
48                    gruppe.PositionAmTisch = positionAmTisch;
49                    gruppe.IstGruppePlatziert = true;
50                    break;
51            }
52        }
53
54        public override string ToString()
55        {
56            return "Einfacher Ober";
57        }
58
59        public virtual void GruppeGeht(IGruppe gruppe)
60        {
61            // Plaetze als "frei" markieren
62            for (int i = gruppe.PositionAmTisch.GruppePositionStart; i < gruppe.PositionAmTisch.GruppePositionEnde + 1; i++)
63            {
64                _meinTisch.MachePlatzFrei(i);
65            }
66        }
67

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 2
68     public virtual GruppePositionAmTisch GruppeKommt(IGruppe gruppe)
69     {
70         // Suche ein freies Platzintervall, das moeglichst genau so gross wie die ↵
71         // Gruppe ist.
72         int startPosition = Algorithmen.SubarrayMitMindestlaenge(_meinTisch. ↵
73             IstPlatzFrei, gruppe.Personen.Count, ↵
74             _meinTisch.Tischgroesse);
75         var position = new GruppePositionAmTisch();
76
77         if (startPosition == -1)
78         {
79             // Kein freier Platz am Tisch, die Gruppe wird NICHT platziert. Der ↵
80             // Ober aergert sich erst dann, wenn die Gruppe nach Hause geht (weil sie nicht ↵
81             // platziert wurde).
82             position.GruppePositionStart = position.GruppePositionEnde = -1;
83             position.GruppePositionStart = -1;
84             return position;
85         }
86         else
87         {
88             // Es wurde ein passendes Platzintervall gefunden
89             position.GruppePositionStart = startPosition;
90             position.GruppePositionEnde = startPosition + gruppe.Personen.Count - 1 ↵
91 ;
92             position.GruppeGroesse = gruppe.Personen.Count;
93             position.GruppeTisch = _meinTisch;
94
95             // Beim Chefober nachfragen, ob das so in Ordnung ist
96             if (!_meinRestaurant.ChefOber.ErlaubnisEinholen(OberAktion. ↵
97                 GruppePlatzieren, new List<int>(), gruppe, position))
98             {
99                 // Der Chefober ist damit nicht einverstanden, auf Delegierung ↵
100                warten!
101                // (Der "einfache Chefober" ist IMMER einverstanden!)
102                position.GruppePositionStart = position.GruppePositionEnde = -2;
103                return position;
104            }
105            gruppe.PositionAmTisch = position;
106            gruppe.IstGruppePlatziert = true;
107
108            // Markiere das entsprechende Intervall am Tisch als belegt
109            for (int platz = position.GruppePositionStart; platz < position. ↵
110                GruppePositionEnde + 1; platz++)
111            {
112                _meinTisch.SetSitzplaetze(platz, gruppe);
113            }
114
115            return position;
116        }
117
118        public virtual void SimulationStart()
119        {
120        }
121
122        public virtual void GuiEreignisseCallBackZeigerLoeschen()
123        {
124        }
125
126        public virtual void GuiEreignisseSetzen()
127        {
128        }
129        public virtual void SimulationPause()
130        {

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 3

```

130
131         }
132
133     public virtual void SimulationStopp()
134     {
135
136     }
137
138     public virtual SteuerelementEinstellungen GuiEinstellungen()
139     {
140         var fenster = new EinfacherOberEinstellungen();
141         fenster.DasObjekt = this;
142         return fenster;
143     }
144 }
145
146 [Serializable]
147 public class EinfacherChefOber : IChefOber
148 {
149     protected Dequeue<IGruppe> _gruppenSchlange;
150     protected TickEreignis _tickEreignis;
151     protected Restaurant _meinRestaurant;
152
153     public Restaurant DasRestaurant
154     {
155         get
156         {
157             return _meinRestaurant;
158         }
159
160         set
161         {
162             EntferneTickEreignis();
163             _meinRestaurant = value;
164             _tickEreignis = new TickEreignis(OnTick);
165             _meinRestaurant.Tick += _tickEreignis;
166         }
167     }
168
169     private void EntferneTickEreignis()
170     {
171         try
172         {
173             _meinRestaurant.Tick -= _tickEreignis;
174         }
175         catch
176         {
177
178         }
179     }
180
181     public bool ErlaubnisEinholen(OberAktion aktion, List<int> parameter, IGruppe gruppe, GruppePositionAmTisch positionAmTisch)
182     {
183         return true;
184     }
185
186     public void GruppeGeht(IGruppe gruppe)
187     {
188         // Leite die Information an den zustaendigen Ober weiter
189         gruppe.PositionAmTisch.GruppeTisch.Ober.GruppeGeht(gruppe);
190     }
191
192     public GruppePositionAmTisch GruppeKommt(IGruppe gruppe)
193     {
194         // Fuege Gruppe zur Schlange hinzu (am Ende)
195         _gruppenSchlange.EinfuegenEnde(gruppe);
196         return default(GruppePositionAmTisch);
197     }
198 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 4
199     /// <summary>
200     /// Wird aufgerufen, wenn das Tick-Ereignis ausgelöst wird.
201     /// </summary>
202     /// <param name="uhrzeit">Die aktuelle Uhrzeit (Anzahl vergangener
203     Zeiteinheiten)</param>
204     public virtual void OnTick(long uhrzeit)
205     {
206         int anzahlGruppen = _gruppenSchlange.Groesse();
207
208         // Platziere alle freien Gruppen
209         for (int i = 0; i < anzahlGruppen; i++)
210         {
211             var aktuelleGruppe = (IGruppe)_gruppenSchlange.EntferneAnfang();
212             DebugNachrichtAnzeigen("Versuche, Gruppe " + aktuelleGruppe.ToString() +
213             " zu platzieren.");
214
215             if (!aktuelleGruppe.IstGruppeSchonGegangen())
216             {
217                 // Die Gruppe ist noch da
218                 // Der Ober von Tisch 0 soll die Gruppe platzieren
219                 GruppePositionAmTisch position = _meinRestaurant.ListeDerTische[0].Ober.GruppeKommt(aktuelleGruppe);
220
221                 if (position.GruppePositionStart < 0)
222                 {
223                     // Kein Platz am Tisch 0 gefunden
224                     // Das "einfache Restaurant" hat keine weiteren Tische oder
225                     Optimierungen, also Pech gehabt!
226                     _gruppenSchlange.EinfuegenEnde(aktuelleGruppe);
227                     DebugNachrichtAnzeigen("Es wurde keine passende Position
228                     gefunden.");
229                 }
230                 else
231                 {
232                     DebugNachrichtAnzeigen("Gruppe positioniert bei [" +
233                     GruppePositionStart.ToString() +
234                     "; " + position.GruppePositionEnde.
235                     ToString() + "]");
236                 }
237             }
238             _meinRestaurant.ListeDerTische[0].EreignisAusloesenGUINeuZeichnen();
239         }
240
241         public EinfacherChefOber(Restaurant restaurant)
242         {
243             KonstruktorEinArgument(restaurant);
244
245             protected void KonstruktorEinArgument(Restaurant restaurant)
246             {
247                 _meinRestaurant = restaurant;
248                 _gruppenSchlange = new Dequeue<IGruppe>();
249                 _tickEreignis = new TickEreignis(OnTick);
250                 _meinRestaurant.Tick += _tickEreignis;
251
252                 /// <summary>
253                 /// Achtung: Nur fuer Serialisierung verwenden!
254                 /// </summary>
255                 public EinfacherChefOber()
256                 {
257
258                     ~EinfacherChefOber()
259                 {
260                     EntferneTickEreignis();
261                 }

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs

5

```
262     public override string ToString()
263     {
264         return "Einfacher Chef-Ober";
265     }
266
267     public void SimulationStart()
268     {
269
270     }
271
272     public void GuiEreignisseCallBackZeigerLoeschen()
273     {
274
275     }
276
277     public void GuiEreignisseSetzen()
278     {
279
280     }
281
282     public void SimulationPause()
283     {
284
285     }
286
287     public void SimulationStopp()
288     {
289
290     }
291
292     public SteuerelementEinstellungen GuiEinstellungen()
293     {
294         var fenster = new EinfacherChefOberEinstellungen();
295         fenster.DasObjekt = this;
296         return fenster;
297     }
298
299
300     private void DebugNachrichtAnzeigen(string text)
301     {
302         StatischeObjekte.DebugNachricht("[Einfacher Chefober] " + text);
303     }
304 }
305
306 [Serializable]
307 public class EinfacheGruppe : IGruppe
308 {
309     protected GruppePositionAmTisch _positionAmTisch;
310     protected int _bisherigeAufenthaltsdauer;
311     protected int _wartezeitsitzplatz;
312     protected bool _istGruppeSchonGegangen;
313     protected bool _istGruppeSchonPlatziert;
314     protected Restaurant _meinRestaurant;
315     protected List<IPerson> _personen;
316     protected TickEreignis _tickEreignis;
317     protected System.Drawing.Color _guiMeineFarbe = GUI.GuiDesign.NaechsteFarbe();
318     private int _meinIndex = GruppenIndizes.NaechsterIndex();
319
320     public virtual System.Drawing.Color GuiGruppenFarbe
321     {
322         get
323         {
324             return _guiMeineFarbe;
325         }
326
327         set
328         {
329             _guiMeineFarbe = value;
330         }
331     }
```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 6

```
332     public GruppePositionAmTisch PositionAmTisch
333     {
334         get
335         {
336             return _positionAmTisch;
337         }
338         set
339         {
340             _positionAmTisch = value;
341         }
342     }
343
344
345
346     public bool IstGruppePlatziert
347     {
348         get
349         {
350             return _istGruppeSchonPlatziert;
351         }
352         set
353         {
354             _istGruppeSchonPlatziert = value;
355         }
356     }
357
358
359     public Restaurant DasRestaurant
360     {
361         get
362         {
363             return _meinRestaurant;
364         }
365         set
366         {
367             EntferneTickEreignis();
368             _meinRestaurant = value;
369             _tickEreignis = new TickEreignis(OnTick);
370             _meinRestaurant.Tick += _tickEreignis;
371         }
372     }
373
374
375     public List<IPerson> Personen
376     {
377         get
378         {
379             return _personen;
380         }
381         set
382         {
383             _personen = value;
384         }
385     }
386
387
388     private void EntferneTickEreignis()
389     {
390         try
391         {
392             _meinRestaurant.Tick -= _tickEreignis;
393         }
394         catch
395         {
396
397         }
398     }
399
400     public int CompareTo(object obj)
401     {
```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 7
402         int wartezeitObj = ((IGruppe)obj).WartezeitSitzplatz();
403         return wartezeitObj - _wartezeitSitzplatz;
404     }
405
406     public int BisherigeAufenthaltsdauer()
407     {
408         return _bisherigeAufenthaltsdauer;
409     }
410
411     public int WartezeitSitzplatz()
412     {
413         return _wartezeitSitzplatz;
414     }
415
416     public bool IstGruppeSchonGegangen()
417     {
418         return _istGruppeSchonGegangen;
419     }
420
421     public virtual bool IstSchuelerGruppe()
422     {
423         return true;
424     }
425
426     public int EindeutigerIndex()
427     {
428         return _meinIndex;
429     }
430
431     public EinfacheGruppe()
432     {
433         // Variablen initialisieren
434         _positionAmTisch = new GruppePositionAmTisch();
435         _positionAmTisch.GruppePositionStart = -1;
436         _bisherigeAufenthaltsdauer = 0;
437         _wartezeitSitzplatz = 2;
438         _istGruppeSchonPlatziert = false;
439         _istGruppeSchonGegangen = false;
440         _personen = new List<IPerson>();
441     }
442
443     public EinfacheGruppe(Restaurant restaurant, int anzahlDerPersonen)
444     {
445         // Variablen initialisieren
446         _positionAmTisch = new GruppePositionAmTisch();
447         _positionAmTisch.GruppePositionStart = -1;
448         _bisherigeAufenthaltsdauer = 0;
449         _wartezeitSitzplatz = 2;
450         _istGruppeSchonPlatziert = false;
451         _istGruppeSchonGegangen = false;
452         _personen = new List<IPerson>();
453
454         for (int i = 0; i < anzahlDerPersonen; i++)
455         {
456             // Erstelle Personen
457             _personen.Add(new Person("KEIN NAME"));
458         }
459
460         DasRestaurant = restaurant;
461     }
462
463     /// <summary>
464     /// Wird aufgerufen, wenn das Tick-Ereignis ausgelöst wird.
465     /// </summary>
466     /// <param name="uhrzeit">Die aktuelle Uhrzeit (Anzahl vergangener
467     Zeiteinheiten)</param>
468     public void OnTick(long uhrzeit)
469     {
470         _bisherigeAufenthaltsdauer++;

```



```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 8
471         if (uhrzeit > _wartezzeitSitzplatz && !IstGruppePlatziert)
472         {
473             // Der Ober aergert sich, die Gruppe geht
474             _meinRestaurant.EreignisAusloesenOberAergertSich(default(IOber),
475             _meinRestaurant.ChefOber, this,
476             new List<int>());
477             GruppeAufloesen();
478         }
479     }
480     ~EinfacheGruppe()
481     {
482         EntferneTickEreignis();
483     }
484
485     public void GruppeAufloesen()
486     {
487         EntferneTickEreignis();
488         _istGruppeSchonGegangen = true;
489     }
490
491     public override string ToString()
492     {
493         return "|Einfache Gruppe|=" + this.Personen.Count.ToString();
494     }
495
496     public void SimulationStart()
497     {
498
499     }
500
501     public void GuiEreignisseCallBackZeigerLoeschen()
502     {
503
504     }
505
506     public void GuiEreignisseSetzen()
507     {
508
509     }
510
511     public void SimulationPause()
512     {
513
514     }
515
516     public void SimulationStopp()
517     {
518
519     }
520
521     public SteuerelementEinstellungen GuiEinstellungen()
522     {
523         var fenster = new EinfacheGruppeEinstellungen();
524         fenster.DasObjekt = this;
525         return fenster;
526     }
527 }
528
529 [Serializable]
530 public class EinfacherKoordinator : IGruppenKoordinator
531 {
532     protected Restaurant _meinRestaurant;
533     private object __listeDerGruppen;
534     protected int _anzahlDerPersonen;
535     protected int _internerZustand;
536     protected TickEreignis _tickEreignis;
537     protected IGruppe _letzteErstelleGruppe = default(IGruppe);
538     protected int[] _intervallFolge;
539     protected bool _gruppenMitUebrigenPersonenAuffuellen;

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 9
540     protected IObjectContainer<IPerson> FreiePersonen;
541     protected int _personenAusblenden = 0;
542
543     public int PersonenAusblenden
544     {
545         get
546         {
547             return _personenAusblenden;
548         }
549         set
550         {
551             _personenAusblenden = value;
552             if (FreiePersonen != default.IContainerControl)) FreiePersonen.
553             PersonenAusblenden = value;
554         }
555     }
556
557     public List<IGruppe> _listeDerGruppen
558     {
559         get
560         {
561             return (List<IGruppe>) __listeDerGruppen;
562         }
563         set
564         {
565             __listeDerGruppen = value;
566         }
567     }
568
569     public int[] IntervallFolge
570     {
571         get
572         {
573             return _intervallFolge;
574         }
575         set
576         {
577             _intervallFolge = value;
578         }
579     }
580
581     public bool GruppenMitUebrigenPersonenAuffuellen
582     {
583         get
584         {
585             return _gruppenMitUebrigenPersonenAuffuellen;
586         }
587         set
588         {
589             _gruppenMitUebrigenPersonenAuffuellen = value;
590         }
591     }
592 }
593
594     public bool ErlaubnisEinholen(GruppeAktion aktion, IGruppe gruppe, List<int> parameters)
595     {
596         // Ober ist immer einverstanden
597         if (aktion == GruppeAktion.TischVerlassen)
598         {
599             // Gruppe aus der Liste der Gruppen loeschen
600             _listeDerGruppen.Remove(gruppe);
601         }
602
603         return true;
604     }
605
606     public Restaurant DasRestaurant
607     {

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 10
608         get
609     {
610         return _meinRestaurant;
611     }
612
613     set
614     {
615         EntferneTickEreignis();
616         _meinRestaurant = value;
617         _tickEreignis = new TickEreignis(OnTick);
618         _meinRestaurant.Tick += _tickEreignis;
619     }
620 }
621
622     public virtual int AnzahlDerPersonen
623     {
624         get
625     {
626         return _anzahlDerPersonen;
627     }
628
629         set
630     {
631         // Freie Personen erstellen
632         FreiePersonen = new ObjectList<IPerson>();
633         FreiePersonen.PersonenAusblenden = _personenAusblenden;
634         _listeDerGruppen = new List<IGruppe>();
635
636         _anzahlDerPersonen = value;
637
638         for (int i = 0; i < _anzahlDerPersonen; i++)
639         {
640             FreiePersonen.FuegeElementEin(new Person("Person " + i.ToString()));
641         }
642     }
643 }
644
645 /// <summary>
646 /// Wird aufgerufen, wenn das Tick-Ereignis ausgelöst wird.
647 /// </summary>
648 /// <param name="uhrzeit">Die aktuelle Uhrzeit (Anzahl vergangener
Zeiteinheiten)</param>
649     public virtual void OnTick(long uhrzeit)
650     {
651         OnTickEreignisFuerTisch(0, FreiePersonen.Count());
652     }
653
654     protected void OnTickEreignisFuerTisch(int tischIndex, int
param_gesamtAnzahlFreiePersonen)
655     {
656         int gesamtAnzahlFreiePersonen = param_gesamtAnzahlFreiePersonen;
657
658         if (_letzteErstelleGruppe == default(IGruppe) || _letzteErstelleGruppe.
IstGruppePlatziert)
659         {
660
661             StatischeObjekte.DebugNachricht("[Tick-Ereignis beim einfachen
Koordinator] " +
gesamtAnzahlFreiePersonen.ToString() +
" freie Personen.");
662
663             // Versuche, die freien zusammenhaengenden Plaetze zu "zerstueckeln"
664             if (_internerZustand == 0)
665             {
666                 StatischeObjekte.DebugNachricht(
"- interner Zustand = 0, weise jeder Person eine eigene Gruppe
zu.");
667             }
668         }
669     }
670 }
671

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 11
672          // Weise jeder Person eine eigene Gruppe zu
673          int anzPersonen = gesamtAnzahlFreiePersonen;
674          for (int i = 0; i < anzPersonen; i++)
675          {
676              EinfacheGruppe neueGruppe = ErstelleNeueGruppe(1);
677              _listeDerGruppen.Add(neueGruppe);
678              _meinRestaurant.ChefOber.GruppeKommt(neueGruppe);
679              _letzteErstelleGruppe = neueGruppe;
680          }
681          _internerZustand++;
682      }
683      else
684      {
685          StatischeObjekte.DebugNachricht("- interner Zustand = " +
686                                         _internerZustand.ToString() + ".");
687
688          if (_internerZustand > _intervallFolge.Length ||
689              _meinRestaurant.ListeDerTische[tischIndex].Tischgroesse <
690              _intervallFolge[_internerZustand - 1])
691          {
692              // Es soll ein Platz freigemacht werden, der groesser als der
693              Tisch ist, wenn der Algorithmus jetzt noch nicht gewonnen hat, dann wird er das ↵
694              auch nicht mehr!
695              // Oder: Die Folge ist zu Ende
696              _meinRestaurant.EreignisAusloesenKoordinatorAufgeben(this);
697              StatischeObjekte.DebugNachricht("Der Koordinator gibt auf!");
698              StatischeObjekte.DebugNachricht(
699                  "[Tick-Ereignis beim einfachen Koordinator] Ende des Tick- ↵
700 Ereignis.");
701          }
702          return;
703      }
704
705      StatischeObjekte.DebugNachricht(
706          "Mache Platz frei, Lueckengroesse maximal " +
707          (_intervallFolge[_internerZustand - 1] - 1).ToString() + ".");
708
709      // Mache Platz fuer groessere Gruppen
710      List<IGruppe> zuLoeschendeGruppen =
711          Algorithmen.EinfachesRestaurant_EntferneGruppen(
712              _meinRestaurant.ListeDerTische[tischIndex],
713              _intervallFolge[_internerZustand - 1] - 1);
714
715      StatischeObjekte.DebugNachricht(
716          "Der Algorithmus schlaegt vor, die Gruppen " +
717          DebugNachrichtListeGruppe(zuLoeschendeGruppen) + " zu loeschen. ↵
718 ");
719
720      System.Threading.Thread.Sleep(_meinRestaurant.Intervall);
721
722      // Loesche die Gruppen nun
723      foreach (IGruppe gruppe in zuLoeschendeGruppen)
724      {
725          gruppe.GruppeAufloesen();
726          if (!_listeDerGruppen.Remove(gruppe))
727          {
728              // Gruppe konnte nicht aus der "Liste aller Gruppen" ↵
729              entfernt werden! Das darf nicht passieren!
730              StatischeObjekte.DebugNachricht("!!!ERR Gruppe " + gruppe. ↵
731
732              ToString() +
733              " konnte nicht aus der Liste aller Gruppen entfernt werden.");
734          }
735      }
736
737      foreach (IPerson person in gruppe.Personen)
738      {
739          FreiePersonHinzufuegen(person);
740      }
741
742      _meinRestaurant.ChefOber.GruppeGeht(gruppe);
743  }

```



```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 13
796                               anzahlFreiePersonen -
797                               (anzahlDerGruppen *
798                               _intervallFolge[_internerZustand - 1]);
799                               EinfacheGruppe neueGruppe = ErstelleNeueGruppe(groesse) ↵
800 ;
801                               _listeDerGruppen.Add(neueGruppe);
802                               _meinRestaurant.ChefOber.GruppeKommt(neueGruppe);
803                               _letzteErstelleGruppe = neueGruppe;
804 }
805 {
806     // Die letzte Gruppe wird nicht groesser
807     EinfacheGruppe neueGruppe =
808         ErstelleNeueGruppe(_intervallFolge[_internerZustand ↵
809         - 1]);
810         _listeDerGruppen.Add(neueGruppe);
811         _meinRestaurant.ChefOber.GruppeKommt(neueGruppe);
812         _letzteErstelleGruppe = neueGruppe;
813     }
814 }
815
816     StatischeObjekte.DebugNachricht("Es wurden " + anzahlDerGruppen ↵
817     .ToString() +
818     " Gruppen erstellt.");
819 }
820     _internerZustand++;
821 }
822
823     StatischeObjekte.DebugNachricht(
824         "[Tick-Ereignis beim einfachen Koordinator] Ende des Tick-Ereignis. ↵
825     ");
826 }
827     StatischeObjekte.DebugNachricht(
828         "Es wurden noch nicht alle Gruppen platziert, warten..."); ↵
829 }
830 }
831
832     protected virtual void FreiePersonHinzufuegen(IPerson person)
833     {
834         FreiePersonen.FuegeElementEin(person);
835     }
836
837     protected virtual IPerson FreiePersonHolen()
838     {
839         IPerson rueckgabe = FreiePersonen.HoleErstesElement();
840
841         return rueckgabe;
842     }
843
844     private string DebugNachrichtListeGruppe(List<IGruppe> gruppen)
845     {
846         string rueckgabe = "[";
847
848         for (int i = 0; i < gruppen.Count; i++)
849         {
850             if (i == gruppen.Count - 1)
851             {
852                 rueckgabe += "[" + gruppen[i].PositionAmTisch.GruppePositionStart. ↵
853                 ToString() + ";" + gruppen[i].PositionAmTisch.GruppePositionEnde.ToString() ↵
854                 () + "]";
855             }
856             else
857             {
858                 rueckgabe += "[" + gruppen[i].PositionAmTisch.GruppePositionStart. ↵
859                 ToString() + ";" + gruppen[i].PositionAmTisch.GruppePositionEnde.ToString() ↵

```

```

F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 14
    () + " ] " + " ; " ;
859        }
860    }
861    rueckgabe += " } ";
862    return rueckgabe;
863 }
864 }
865 /**
866 * <summary>
867 * Erstellt eine neue einfache Gruppe, die eine bestimmte Anzahl an Personen
868 * enthaelt.
869 * </summary>
870 * <param name="anzahlDerPersonen">Anzahl der Personen</param>
871 * <returns>Zeiger auf die Gruppe</returns>
872 public virtual EinfacheGruppe ErstelleNeueGruppe(int anzahlDerPersonen)
873 {
874     EinfacheGruppe neueGruppe = new EinfacheGruppe();
875
876     for (int i = 0; i < anzahlDerPersonen; i++)
877     {
878         neueGruppe.Personen.Add(FreiePersonHolen());
879     }
880
881     neueGruppe.IstGruppePlatziert = false;
882     neueGruppe.DasRestaurant = _meinRestaurant;
883
884     return neueGruppe;
885 }
886
887 public EinfacherKoordinator(Restaurant restaurant)
888 {
889     KonstruktorEinArgument(restaurant);
890 }
891
892 protected void KonstruktorEinArgument(Restaurant restaurant)
893 {
894     _meinRestaurant = restaurant;
895     _internerZustand = 0;
896     _tickEreignis = new TickEreignis(OnTick);
897     _meinRestaurant.Tick += _tickEreignis;
898
899     _intervallFolge = new int[] { 2, 4, 8, 16, 32 };
900 }
901
902 private void EntferneTickEreignis()
903 {
904     try
905     {
906         _meinRestaurant.Tick -= _tickEreignis;
907     }
908     catch
909     {
910     }
911 }
912
913 ~EinfacherKoordinator()
914 {
915     EntferneTickEreignis();
916 }
917
918 /**
919 * Achtung: Nur fuer Serialisierung verwenden!
920 */
921 public EinfacherKoordinator()
922 {
923 }
924
925 public override string ToString()

```

```
F:\Users\meister\Documents\Visual Studio ...\\EinfachesRestaurant.cs 15
927     {
928         return "Einfacher Koordinator";
929     }
930
931     public void SimulationStart()
932     {
933
934     }
935
936     public void GuiEreignisseCallBackZeigerLoeschen()
937     {
938
939     }
940
941     public void GuiEreignisseSetzen()
942     {
943
944     }
945
946     public void SimulationPause()
947     {
948
949     }
950
951     public void SimulationStopp()
952     {
953
954     }
955
956     public virtual SteuerelementEinstellungen GuiEinstellungen()
957     {
958         var fenster = new EinfacherKoordinatorEinstellungen();
959         fenster.DasObjekt = this;
960         return fenster;
961     }
962 }
963 }
964 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 1
1 using System;
2 using System.Collections.Generic;
3 using System.IO;
4 using Facet.Combinatorics;
5
6 namespace _28_2_bwinf_2.RestaurantAlgorithmen
7 {
8
9     public enum Spieler
10    {
11        Ober = 1, Koordinator = 2, Unbekannt = 0
12    }
13
14    public class MinimaxOber : EinfacherOber
15    {
16        private SpielBaumKnoten _spielbaumWurzel;
17        public int[] _intervallFolge = new int[] { 2, 5, 8, 16, 32 };
18        private bool _gruppenMitUebrigenPersonenAuffuellen;
19        public static int _anzahlSitzplaetze;
20
21        public int AnzahlDerSitzplaetze
22        {
23            get
24            {
25                return _anzahlSitzplaetze;
26            }
27
28            set
29            {
30                _anzahlSitzplaetze = value;
31            }
32        }
33
34        public override string ToString()
35        {
36            return "Perfekter Ober";
37        }
38
39        public List<string> SpielzuegeBerechnen(int freiePersonen)
40        {
41            List<string> rueckgabe = new List<string>();
42
43            // Wurzelknoten erzeugen
44            _spielbaumWurzel = new SpielBaumKnoten();
45            _spielbaumWurzel.Tisch = Algorithmen.EinfachesRestaurant_BerechneKosten
        (_meinTisch, 0);
46            _spielbaumWurzel.FreiePersonen = freiePersonen;
47
48            // Rekursiven Algorithmus starten
49            ZustandGewinner x = ErmittleGewinner(_spielbaumWurzel, Spieler.Koordinator,
        0);
50
51            for (int i = 0; i < x.Spielzuege.Count; i++)
52            {
53                Console.WriteLine(x.Spielzuege[i].ToString());
54                rueckgabe.Add(x.Spielzuege[i].ToString());
55            }
56
57
58            if (x.Gewinner == Spieler.Koordinator)
59            {
60                Console.WriteLine("Koordinator gewinnt.");
61                rueckgabe.Add("Koordinator gewinnt.");
62            }
63            else
64            {
65                Console.WriteLine("Ober gewinnt.");
66                rueckgabe.Add("Ober gewinnt.");
67            }
68

```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 2
69         return rueckgabe;
70     }
71
72     private string SpielerEnumToString(Spieler spieler)
73     {
74         if (spieler == Spieler.Ober)
75         {
76             return "Ober";
77         }
78         else if (spieler == Spieler.Koordinator)
79         {
80             return "Koordinator";
81         }
82         else
83         {
84             return "Unbekannt";
85         }
86     }
87
88     private void DebugNachrichtAusgeben(string text, int rekursionstiefe)
89     {
90         string message = "[PERFEKT] ";
91
92         for (int i = 0; i < rekursionstiefe; i++)
93         {
94             message += "   ";
95         }
96
97         message += text;
98         Console.WriteLine(message);
99
100        StreamWriter datei = new StreamWriter("algo-perfekt.txt", true);
101        datei.Write(message + "\n");
102        datei.Close();
103    }
104
105    public ZustandGewinner ErmittleGewinner(SpielBaumKnoten zustand, Spieler aktuellerSpieler, int rekursionstiefe)
106    {
107        DebugNachrichtAusgeben("ErmittleGewinner mit Zustand " + zustand.ToString() +
108        " fuer Spieler " + SpielerEnumToString(aktuellerSpieler) + " und " +
109        "Rekursionstiefe " + rekursionstiefe.ToString() + " gestartet.", rekursionstiefe);
110
111        if (aktuellerSpieler == Spieler.Koordinator)
112        {
113            // Fuer den Koordinator wird der Algorithmus von "Einfacher Koordinator" angewendet
114            SpielBaumKnoten nachfolger = new SpielBaumKnoten();
115            nachfolger.FreiePersonen = zustand.FreiePersonen;
116            nachfolger.Rekursionstiefe = rekursionstiefe + 1;
117
118            // internen Zustand berechnen: ergibt sich aus rekursionstiefe
119            int _internerZustand = rekursionstiefe / 2;
120
121            if (_internerZustand == 0)
122            {
123                // Generiere Gruppen der Groesse 1
124                nachfolger.ZuPlatzierendeGruppen = new List<int>();
125
126                for (int i = 0; i < zustand.FreiePersonen; i++)
127                {
128                    nachfolger.ZuPlatzierendeGruppen.Add(1);
129                }
130
131                nachfolger.FreiePersonen = 0;
132                nachfolger.Tisch = Algorithmen.EinfachesRestaurant_KopiereStruktur(zustand.Tisch);

```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 3
133         }
134     else
135     {
136         if (_internerZustand > _intervallFolge.Length || 
137             zustand.Tisch.Plaetze.Length < _intervallFolge[_internerZustand - 1])
138         {
139             // Abbruchbedingung der Rekursion: Der Koordinator gibt auf
140             (genaue Erklaerung siehe EinfacherKoordinator)
141             DebugNachrichtAusgeben("Koordinator gibt auf.", 
142             rekursionstiefe);
143
144             ZustandGewinner rueckgabe = new ZustandGewinner();
145             rueckgabe.Gewinner = Spieler.Ober;
146             rueckgabe.Spielzuege = new List<SpielBaumKnoten>();
147             rueckgabe.Spielzuege.Add(zustand);
148             return rueckgabe;
149
150             // Jetzt Gruppen entfernen
151             zustand.Tisch.MaximaleLueckenGroesse = _intervallFolge[_internerZustand - 1] - 1;
152             nachfolger.Tisch = Algorithmen.EinfachesRestaurant_EntferneGruppen(
153                 (zustand.Tisch,
154                  _intervallFolge[_internerZustand - 1] - 1);
155
156             // Indizes der Gruppen in der Tisch-Kosten-Struktur anpassen, damit
157             // der Algorithmus auch mit "normalen" Personen funktioniert. Diese haben negative
158             // Indizes und der Algorithmus markiert feste Gruppen auch mit negativen Indizes!
159             for (int i = 0; i < zustand.Tisch.Plaetze.Length; i++)
160             {
161                 if (nachfolger.Tisch.Plaetze[i] < 0 && zustand.Tisch.Plaetze[i] > 0)
162                     {
163                         // Netter Nebeneffekt: Die originalen Indizes bleiben erhalten
164                         nachfolger.Tisch.Plaetze[i] = zustand.Tisch.Plaetze[i];
165                     }
166                     else if (nachfolger.Tisch.Plaetze[i] == 0 && zustand.Tisch.Plaetze[i] > 0)
167                     {
168                         // Untersuche, wie viele Personen entfernt wurden
169                         nachfolger.FreiePersonen++;
170                     }
171
172             // Pruefe, ob genuegend freie Personen vorhanden sind, um den Ober
173             // gleich zu aergern
174             SubarrayInformation groessteZusammenhaengendePlaetze =
175                 Algorithmen.LaengstesSubarray(nachfolger.Tisch.IstPlatzFrei,
176                 nachfolger.Tisch.Plaetze.Length);
177
178             if (nachfolger.FreiePersonen > groessteZusammenhaengendePlaetze.Wert)
179             {
180                 // Es sind genuegend Personen vorhanden, um den Ober jetzt zu
181                 // aergern
182                 ZustandGewinner rueckgabe = new ZustandGewinner();
183                 rueckgabe.Gewinner = Spieler.Koordinator;
184                 rueckgabe.Spielzuege = new List<SpielBaumKnoten>();
185                 rueckgabe.Spielzuege.Add(zustand);
186                 return rueckgabe;
187             }
188             else
189             {
190                 // Erstelle neue Gruppen
191                 int anzahlDerGruppen = nachfolger.FreiePersonen / _intervallFolge[_internerZustand - 1];
192                 int anzahlFreiePersonen = nachfolger.FreiePersonen;

```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 4
188         nachfolger.ZuPlatzierendeGruppen = new List<int>();
189
190         for (int i = 0; i < anzahlDerGruppen - 1; i++)
191         {
192             nachfolger.ZuPlatzierendeGruppen.Add(_intervallFolge[_internerZustand - 1]);
193             nachfolger.FreiePersonen -= _intervallFolge[_internerZustand - 1];
194         }
195
196         if (anzahlDerGruppen > 0)
197         {
198             if (_gruppenMitUebrigenPersonenAuffuellen)
199             {
200                 // Letzte Grupp evtl. groesser machen
201                 int groesse = _intervallFolge[_internerZustand - 1] +
202                     anzahlFreiePersonen -
203                     (anzahlDerGruppen *
204                         _intervallFolge[_internerZustand - 1]);
205                 nachfolger.ZuPlatzierendeGruppen.Add(groesse);
206                 nachfolger.FreiePersonen -= groesse;
207
208                 // Nur zur Sicherheit
209                 if (nachfolger.FreiePersonen != 0)
210                     throw new Exception("Es gibt immer noch freie
211                     Personen!");
212             }
213             else
214             {
215                 nachfolger.ZuPlatzierendeGruppen.Add(_intervallFolge[_internerZustand - 1]);
216                 nachfolger.FreiePersonen -= _intervallFolge[_internerZustand - 1];
217             }
218         }
219     }
220
221     DebugNachrichtAusgeben(
222         "Rekursive Aufruf vorbereitet fuer Ober mit Zustand " + nachfolger.
223         ToString() + ".", rekursionstiefe);
224
225     ZustandGewinner rekursiv = ErmittleGewinner(nachfolger, Spieler.Ober,
226         rekursionstiefe + 1);
227     ZustandGewinner rueckgabe2 = new ZustandGewinner();
228     rueckgabe2.Spielzuege = new List<SpielBaumKnoten>();
229     rueckgabe2.Gewinner = rekursiv.Gewinner;
230     rueckgabe2.Spielzuege.AddRange(rekursiv.Spielzuege);
231     rueckgabe2.Spielzuege.Insert(0, zustand);
232     return rueckgabe2;
233 }
234 else
235 {
236     // Der Ober ist am Zug, generiere alle gueltigen Spielzuege
237
238     // Generiere alle moeglichen Spielzuege
239     // Erstelle Liste mit allen freien Plaetzen
240     List<int> startPos = new List<int>();
241     for (int i = 0; i < zustand.Tisch.Plaetze.Length; i++)
242     {
243         if (zustand.Tisch.Plaetze[i] == 0) startPos.Add(i);
244     }
245
246     // Generiere alle Variationen: Nehme <Anzahl der Gruppen> viele,
247     beliebige <freie Plaetze>, ohne Wiederholungen und mit Beachtung der Reihenfolge
248     Combinations<int> spielzuegeStartPos = new Combinations<int>(startPos,
249     zustand.ZuPlatzierendeGruppen.Count);
250
251     foreach (IList<int> spielzug in spielzuegeStartPos)

```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 5
250
251         {
252             //DebugNachrichtAusgeben("Naechster Spielzug " + spielzug[0].
253             ToString() + "," + spielzug[1].ToString() + "," + spielzug[2].ToString(),
254             rekursionstiefe);
255             bool aktuellerFallIstUngueltig = false;
256             // Ueberpruefe, ob der Spielzug gueltig ist
257
258             // Dieses Array steht auf "wahr", wenn ein Sitzplatz besetzt ist
259             bool[] sitzplatzBesetzt = new bool[zustand.Tisch.Plaetze.Length];
260
261             // Uebertrage Daten aus der Tisch-Kosten-Struktur, also schreibe in
262             das Array, wenn ein Platz schon besetzt ist
263             for (int i = 0; i < zustand.Tisch.Plaetze.Length; i++)
264             {
265                 sitzplatzBesetzt[i] = !zustand.Tisch.IstPlatzFrei(i);
266             }
267
268             // Iteriere ueber alle zu platzierenden Gruppen und besetzte die
269             Plaetze entsprechend
270             for (int i = 0; i < zustand.ZuPlatzierendeGruppen.Count; i++)
271             {
272                 // Iteriere ueber jeden benoetigten Sitzplatz der aktuellen
273                 Gruppe
274                 for (int j = 0; j < zustand.ZuPlatzierendeGruppen[i]; j++)
275                 {
276                     // spielzug[i] ist der erste Sitzplatz der Gruppe i
277                     if ((spielzug[i] + j) >= sitzplatzBesetzt.Length || 
278                         sitzplatzBesetzt[spielzug[i] + j])
279                     {
280                         // Ein Platz, den die Gruppe bekommen soll, ist bereits
281                         besetzt
282                         aktuellerFallIstUngueltig = true;
283                     }
284                     else
285                     {
286                         sitzplatzBesetzt[spielzug[i] + j] = true;
287                     }
288
289                     if (aktuellerFallIstUngueltig) break;
290                 }
291
292                 if (aktuellerFallIstUngueltig) break;
293             }
294
295             if (!aktuellerFallIstUngueltig)
296             {
297                 // Dieser Spielzug ist so gueltig!
298                 SpielBaumKnoten rekursiverAufrufKnoten = new SpielBaumKnoten();
299                 rekursiverAufrufKnoten.FreiePersonen = zustand.FreiePersonen;
300                 rekursiverAufrufKnoten.Tisch = new int[zustand.Tisch.
301                 Plaetze.Length];
302
303                 rekursiverAufrufKnoten.Rekursionstiefe = rekursionstiefe + 1;
304
305                 int maxGruppenIndex = 0;
306                 // Sitzplatzarray kopieren
307                 for (int i = 0; i < zustand.Tisch.Plaetze.Length; i++)
308                 {
309                     rekursiverAufrufKnoten.Tisch.Plaetze[i] = zustand.Tisch.
310                     Plaetze[i];
311                     maxGruppenIndex = Math.Max(zustand.Tisch.Plaetze[i],
312                     maxGruppenIndex);
313                 }
314
315                 // Gruppen platzieren
316                 // Iteriere ueber alle zu platzierenden Gruppen und besetzte
317                 die Plaetze entsprechend
318                 for (int i = 0; i < zustand.ZuPlatzierendeGruppen.Count; i++)
319                 {
320                     maxGruppenIndex++;
321                 }
322             }
323         }
324     }
325 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 6
309             Gruppe                                // Iteriere ueber jeden benoetigten Sitzplatz der aktuellen ↵
310         {
311             for (int j = 0; j < zustand.ZuPlatzierendeGruppen[i]; j++)
312             {
313                 rekursiverAufrufKnoten.Tisch.Plaetze[spielzug[i] + j] = ↵
314             }
315         }
316     }
317     ZustandGewinner rekursiv = ErmittleGewinner
318     (rekursiverAufrufKnoten, Spieler.Koordinator,
319      rekursionstiefe + 1);
320     if (rekursiv.Gewinner == Spieler.Ober)
321     {
322         // Es wurde ein Spielzug gefunden, mit dem der Ober
323         // gewonnen hat, es muss also nicht weiter gesucht werden
324         // Aktuellen Zustand in der Hashtabelle speichern
325         //_spielZustaende.Add(hashTableElement, Spieler.Ober);
326         DebugNachrichtAusgeben("Ober hat einen siegbringenden
327         Spielzug gefunden.");
328         rekursionstiefe);
329         ZustandGewinner rueckgabe2 = new ZustandGewinner();
330         rueckgabe2.Spielzuege = new List<SpielBaumKnoten>();
331         rueckgabe2.Gewinner = rekursiv.Gewinner;
332         rueckgabe2.Spielzuege.AddRange(rekursiv.Spielzuege);
333         rueckgabe2.Spielzuege.Insert(0, zustand);
334         return rueckgabe2;
335     }
336 }
337 }
338 // Es wurde kein Spielzug gefunden, bei dem der Ober gewinnen kann
339 DebugNachrichtAusgeben("Ober hat KEINEN siegbringenden Spielzug
gefunden.", rekursionstiefe);
340 ZustandGewinner rueckgabe3 = new ZustandGewinner();
341 rueckgabe3.Spielzuege = new List<SpielBaumKnoten>();
342 rueckgabe3.Gewinner = Spieler.Koordinator;
343 rueckgabe3.Spielzuege.Add(zustand);
344 return rueckgabe3;
345 }
346 }
347
348 public struct ZustandGewinner
349 {
350     public Spieler Gewinner;
351     public List<SpielBaumKnoten> Spielzuege;
352 }
353
354 /// <summary>
355 /// Spielzustand fuer schnellen Zugriff auf alle notwendigen Informationen.
356 /// Kann in eine SpielzustandHashTable-Struktur umgewandelt werden.
357 /// </summary>
358 public class SpielBaumKnoten
359 {
360     /// <summary>
361     /// Tisch-Kosten-Struktur, die der Algorithmus EntferneGruppen verarbeiten
362     /// kann.
363     /// </summary>
364     public EinfachesRestaurant_TischKosten Tisch;
365
366     /// <summary>
367     /// Speichert die Anzahl der im aktuellen Spielzustand freien Personen.
368     /// </summary>
369     public int FreiePersonen = 0;
370
371     /// <summary>
372     /// Speichert die vom Ober noch zu platzierenden Gruppen (immer Anzahl der ↵

```

```
F:\Users\meister\Documents\Visual Studio ...\\MinimaxRestaurantOber.cs 7
    Personen).
371     /// </summary>
372     public List<int> ZuPlatzierendeGruppen;
373
374     /// <summary>
375     /// Gibt an, wie weit die Rekursion schon "verschachtelt" ist.
376     /// </summary>
377     public int Rekursionstiefe = 0;
378
379     public override string ToString()
380     {
381         string text = " { " + Tisch.ToString() + "; frei:" + FreiePersonen.
382         ToString() + "; gruppen{ ";
383         if (ZuPlatzierendeGruppen != default(List<int>))
384         {
385             for (int i = 0; i < ZuPlatzierendeGruppen.Count; i++)
386             {
387                 text += ZuPlatzierendeGruppen[i].ToString() + "; ";
388             }
389         }
390         text += "}";
391         return text;
392     }
393 }
394 }
395 }
396 }
397 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 1
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.IO;
5
6 namespace _28_2_bwinf_2.RestaurantAlgorithmen
7 {
8     /// <summary>
9     /// Eine Funktion, die ein Array A darstellt. Wenn IstWahr(i) == true, dann ist A [i] == true.
10    /// </summary>
11    /// <param name="index">Index im Array</param>
12    /// <returns>Wahrheitswert</returns>
13    public delegate bool IstWahr(int index);
14
15    public struct SubarrayInformation
16    {
17        public int StartPosition;
18        public int Wert;
19    }
20
21    public static class GruppenIndizes
22    {
23        private static int _aktuellerIndex = 0;
24
25        public static int NaechsterIndex()
26        {
27            if (_aktuellerIndex == int.MaxValue)
28            {
29                _aktuellerIndex = 1;
30            }
31            else
32            {
33                _aktuellerIndex++;
34            }
35
36            return _aktuellerIndex;
37        }
38    }
39
40    public static class Algorithmen
41    {
42        /// <summary>
43        /// Findet ein Intervall im Array mit dem Wert "wahr", das eine Mindestlaenge besitzt und moeglichst nahe an dieser Mindestlaenge ist.
44        /// </summary>
45        /// <param name="array">Array (via Funktionszeiger)</param>
46        /// <param name="mindestlaenge">Mindestlaenge des Subarrays</param>
47        /// <param name="arrayGroesse">Groesse des Arrays</param>
48        /// <returns>Startindex</returns>
49        public static int SubarrayMitMindestlaenge(IstWahr array, int mindestlaenge,
50 int arrayGroesse)
51        {
52            int aktuellePos = -1, bestePos = -1;
53            int besteLaenge = int.MaxValue;
54
55            if (arrayGroesse < mindestlaenge)
56            {
57                // Array ist zu klein
58                return -1;
59            }
60
61            // Startposition fuer Schleife suchen
62            int startPos = -1;
63            for (int i = 0; i < arrayGroesse; i++)
64            {
65                if (!array(i))
66                {
67                    // Es wurde ein "falscher" Platz gefunden (besetzter Sitzplatz)
68                    startPos = i;
69                }
70            }
71
72            if (startPos != -1)
73            {
74                for (int i = startPos + 1; i < arrayGroesse; i++)
75                {
76                    if (array(i))
77                    {
78                        // Es wurde ein "richtiger" Platz gefunden (leerer Sitzplatz)
79                        bestePos = i;
80                    }
81                }
82            }
83
84            if (bestePos != -1)
85            {
86                for (int i = startPos; i <= bestePos; i++)
87                {
88                    if (!array(i))
89                    {
90                        // Es wurde ein "richtiger" Platz gefunden (leerer Sitzplatz)
91                        besteLaenge = i - startPos;
92                    }
93                }
94            }
95
96            return bestePos;
97        }
98    }
99
100   public static void Main()
101   {
102       // Testcode
103   }
104 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 2
68             break;
69         }
70     }
71
72     if (startPos == -1)
73     {
74         // Das ganze Array ist "wahr"
75         return 0;
76     }
77
78     for (int i = startPos; i < 2 * arrayGroesse + startPos; i++)
79     {
80         if (array(i % arrayGroesse))
81         {
82             // Array ist an dieser Stelle wahr
83             if (aktuellePos == -1)
84             {
85                 if (i > arrayGroesse - 1)
86                 {
87                     // Arraygrenze erreicht
88                     break;
89                 }
90                 else
91                 {
92                     aktuellePos = i;
93                 }
94             }
95         }
96     }
97     else
98     {
99         // Array ist an dieser Stelle nicht wahr
100        if (aktuellePos != -1)
101        {
102            if (i - aktuellePos >= mindestlaenge && i - aktuellePos <
besteLaenge)
103            {
104                // Wert aktualisieren, da bessere Stelle gefunden
105                besteLaenge = i - aktuellePos;
106                bestePos = aktuellePos;
107            }
108
109            // Neuen Anfang suchen
110            aktuellePos = -1;
111        }
112    }
113
114    return bestePos;
115 }
116
117 /// <summary>
118 /// Sucht das (laengenmaessig) groesste Subarray, das aus einem durchgehenden "Wahr-Lauf" besteht.
119 /// </summary>
120 /// <param name="array">Array (via Funktionszeiger)</param>
121 /// <param name="arrayGroesse">Groesse (Laenge) des Arrays</param>
122 /// <returns>Laengster zusammenhaengender "Wahr-Lauf"</returns>
123 public static SubarrayInformation LaengstesSubarray(IstWahr array, int
arrayGroesse)
124 {
125     int besteStartPosition = -1;
126     int aktuelleStartPosition = -1;
127     int besterWert = 0;
128
129     for (int aktuellePos = 0; aktuellePos < 2 * arrayGroesse; aktuellePos++)
130     {
131         if (aktuelleStartPosition == -1)
132         {
133             // Noch kein Startpunkt vorhanden
134

```

```
F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\\Algorithmen.cs

135             if (aktuellePos > arrayGroesse)
136             {
137                 // Es wurde bereits einmal im Kreis gelaufen
138                 goto Fertig;
139             }
140         else
141         {
142             if (array(aktuellePos % arrayGroesse))
143             {
144                 aktuelleStartPos = aktuellePos;
145
146                 if (besteStartPosition == -1)
147                 {
148                     besteStartPosition = aktuellePos;
149                     besterWert = 1;
150                 }
151             }
152         }
153     }
154 else
155 {
156     // Es ist bereits ein Startpunkt vorhanden
157
158     if (array(aktuellePos % arrayGroesse))
159     {
160         // Das Array ist an dieser Stelle "wahr"
161
162         if (aktuellePos - aktuelleStartPos + 1 > besterWert)
163         {
164             // Es wurde ein groesseres Subarray gefunden
165             besteStartPosition = aktuelleStartPos;
166             besterWert = aktuellePos - aktuelleStartPos + 1;
167         }
168     }
169 else
170 {
171     // Der aktuelle "Lauf" ist zu Ende
172     aktuelleStartPos = -1;
173 }
174 }
175 }
176
177 goto Fertig;
178
179 Fertig:
180 if (besterWert > arrayGroesse)
181 {
182     // In diesem Fall ist das ganze Array "wahr"
183     besteStartPosition = 0;
184     besterWert = arrayGroesse;
185 }
186
187 SubarrayInformation rueckgabe = new SubarrayInformation();
188 rueckgabe.StartPosition = besteStartPosition;
189 rueckgabe.Wert = besterWert;
190
191 return rueckgabe;
192 }
193
194 /// <summary>
195 /// Nur fuer Debug-Zwecke: Gibt eine Nachricht im Konsolenfenster aus und
196 schreibt eine Log-Datei.
197 /// </summary>
198 public static void AlgoEntferneDebugNachricht(string message, int
rekursionstiefe)
199 {
200     string text = "";
201     for (int i = 0; i < rekursionstiefe; i++)
202     {
203         text += "    ";
204     }
205 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 4
203         }
204
205         text += message;
206
207         //StatischeObjekte.DebugNachricht("[ALGO-ENTFERNE] " + text);
208
209         //StreamWriter datei = new StreamWriter("algo-entferne.txt", true);
210         //datei.WriteLine(text + "\n");
211         //datei.Close();
212     }
213
214     public static void AlgoEntferneDebugErgebnis(string message)
215     {
216         //StatischeObjekte.DebugNachricht("[ALGO-ENTFERNE ERGEBNIS] " + message. ↵
217         ToString());
218     }
219
220     /// <summary>
221     /// Bool-Array fuer Debugzwecke als string ausgeben.
222     /// </summary>
223     public static string AlgoEntferneBoolArrayDebugNachricht(bool[] array)
224     {
225         string rueckgabe = "{ ";
226
227         for (int i = 0; i < array.Length; i++)
228         {
229             rueckgabe += array[i].ToString();
230
231             if (i < array.Length - 1) rueckgabe += "; ";
232         }
233
234         rueckgabe += "}";
235         return rueckgabe;
236     }
237
238     private static int _gruppenZaehler;
239
240     /// <summary>
241     /// Berechnet aus einer Tischklasse eine Tisch-Kosten-Struktur.
242     /// </summary>
243     public static EinfachesRestaurant_TischKosten ↵
244     EinfachesRestaurant_BerechneKosten(Tisch tisch, int maximaleIntervallGroesse)
245     {
246         EinfachesRestaurant_TischKosten kosten = new ↵
247         EinfachesRestaurant_TischKosten();
248         kosten.MaximaleLueckenGroesse = maximaleIntervallGroesse;
249         kosten.Plaetze = new int[tisch.Tischgroesse];
250
251         //_gruppenZaehler = 0;
252         IGruppe aktuelleGruppe = default(IGruppe);
253
254         // Uebertrage alle Gruppen in die Tisch-Kosten-Struktur
255         for (int i = 0; i < tisch.Tischgroesse; i++)
256         {
257             // Platzarray mit "leer" initialisieren
258             kosten.Plaetze[i] = 0;
259
260             if (!tisch.IstPlatzFrei(i))
261             {
262                 // Dieser Platz ist besetzt
263                 if (tisch.GetSitzplaetze(i) != aktuelleGruppe)
264                 {
265                     // Hier beginnt eine neue Gruppe
266                     //_gruppenZaehler++;
267                     aktuelleGruppe = tisch.GetSitzplaetze(i);
268                 }
269
270                 if (aktuelleGruppe.IstSchuelerGruppe())
271                 {
272                     kosten.Plaetze[i] = aktuelleGruppe.EindeutigerIndex();
273                 }
274             }
275         }
276     }

```



```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 6
327
328
329
330     _gruppenZaehler;
331
332         {
333             // Gruppe darf nicht geloescht werden
334             neueKosten.Plaetze[aktuellePosition] = - negativIndexLinkeGruppe = -_gruppenZaehler;
335         }
336         }
337         else if (neueKosten.Plaetze[aktuellePosition] < 0)
338         {
339             // Die erste Gruppe darf nicht entfernt werden, weil es sich nicht um eine "Schuelergruppe" handelt
340             break;
341         }
342     }
343     else if (aktuelleGruppe != neueKosten.Plaetze
344 [aktuellePosition])
345     {
346         // Jetzt beginnt hier eine neue Gruppe
347         break;
348     }
349     else
350     {
351         indexLinkeGruppe = neueKosten.Plaetze[aktuellePosition];
352
353         // Erster Arrayslot ist eine Gruppe
354         if (i == 1 || i == 2)
355         {
356             // Platz gleich freigeben
357             neueKosten.Plaetze[aktuellePosition] = 0;
358         }
359         else
360         {
361             // Gruppe darf nicht geloescht werden
362             neueKosten.Plaetze[aktuellePosition] = - _gruppenZaehler;
363             negativIndexLinkeGruppe = -_gruppenZaehler;
364         }
365     }
366
367     // Naechster Sitzplatz
368     aktuellePosition++;
369 }
370 }
371
372 {
373     _gruppenZaehler++;
374
375     // Rechte Gruppe am rechten Rand loeschen oder als fest markieren
376     int aktuelleGruppe = neueKosten.Plaetze[neueKosten.Plaetze.Length - 1];
377
378     int aktuellePosition = neueKosten.Plaetze.Length - 1;
379
380     while (aktuellePosition > -1)
381     {
382         if (aktuelleGruppe == 0)
383         {
384             // Am rechten Rand ist mind. ein unbesetzter Platz
385
386             if (neueKosten.Plaetze[aktuellePosition] > 0)
387             {
388                 // Jetzt beginnt die erste Gruppe, diese soll evtl. geloescht werden
389                 aktuelleGruppe = neueKosten.Plaetze[aktuellePosition];

```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 7

390
391                     if (indexLinkeGruppe == aktuelleGruppe)
392                     {
393                         // Es handelt sich um eine Gruppe, die am Anfang ↵
394                         und am Ende des Arrays sitzt ↵
395                         Entscheidung abhaengig! ↵
396
397                         if (i == 1 || i == 2)
398                         {
399                             // Arrayslot einfach freigeben
400                             neueKosten.Plaetze[aktuellePosition] = 0;
401                         }
402                         else
403                         {
404                             neueKosten.Plaetze[aktuellePosition] = ↵
405                             negativIndexLinkeGruppe; ↵
406                         }
407                         else
408                         {
409                             if (i == 1 || i == 3)
410                             {
411                                 // Arrayslot einfach freigeben
412                                 neueKosten.Plaetze[aktuellePosition] = 0;
413                             }
414                             else
415                             {
416                                 neueKosten.Plaetze[aktuellePosition] = - ↵
417                                 _gruppenZaehler; ↵
418                             }
419                         }
420                         else if (neueKosten.Plaetze[aktuellePosition] < 0)
421                         {
422                             // Die erste Gruppe darf nicht entfernt werden, weil ↵
423                             es sich nicht um eine "Schuelergruppe" handelt
424                             break;
425                         }
426                         else if (aktuelleGruppe != neueKosten.Plaetze
427 [aktuellePosition])
428                         {
429                             // Jetzt beginnt hier eine neue Gruppe
430                             break;
431                         }
432                         else if (aktuelleGruppe < 0)
433                         {
434                             // Eine Gruppe mit negativen Index darf nicht entfernt ↵
435                             werden ↵
436                             break;
437                         }
438                         else
439                         {
440                             if (indexLinkeGruppe == neueKosten.Plaetze
441 [aktuellePosition])
442                             {
443                                 // Es handelt sich um eine Gruppe, die am Anfang und ↵
444                                 am Ende des Arrays sitzt ↵
445                                 Entscheidung abhaengig! ↵
446
447                                 if (i == 1 || i == 2)
448                                 {
449                                     // Arrayslot einfach freigeben
450                                     neueKosten.Plaetze[aktuellePosition] = 0;
451                                 }
452                                 else
453                                 {
454                                     neueKosten.Plaetze[aktuellePosition] = ↵

```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 8

        negativIndexLinkeGruppe;
450            }
451        }
452    }
453    {
454        if (i == 1 || i == 3)
455        {
456            // Arrayslot einfach freigeben
457            neueKosten.Plaetze[aktuellePosition] = 0;
458        }
459    }
460    {
461        neueKosten.Plaetze[aktuellePosition] = -
462        _gruppenZaehler;
463    }
464}
465
466        // Naechster Sitzplatz
467        aktuellePosition--;
468    }
469}
470
471        AlgoEntferneDebugNachricht("Tischstruktur fuer Fall " + i.ToString() + ↵
472        ": " + neueKosten.ToString(), 0);
473
474        // Eigentlichen Algorithmus fuer den aktuellen Fall aufrufen
475        EinfachesRestaurant_TischKosten aktuellerFall =
476        EinfachesRestaurant_EntferneGruppen(neueKosten,
477        tischOriginalKosten,
478        1);
479        if (aktuellerFall.Kosten != int.MaxValue)
480        {
481            // Wenn Kosten==int.MaxValue, hat der rekursive Algorithmus keine ↵
482            Loesung gefunden
483            // Pruefe Ergebnis auf Gueltigkeit
484            if (EinfachesRestaurant_PruefeTischStruktur(aktuellerFall,
485            tischOriginalKosten))
486            {
487                int aktuellerFallKosten = EinfachesRestaurant_BerechneKosten ↵
488                (aktuellerFall);
489
490                AlgoEntferneDebugNachricht(
491                    "Aktueller Fall ist gueltig mit Kosten " +
492                    aktuellerFallKosten.ToString() + " und Ergebnis " +
493                    aktuellerFall.ToString() + ".\n\n", 0);
494
495                if (aktuellerFallKosten < kostenBesteEntscheidung)
496                {
497                    kostenBesteEntscheidung = aktuellerFallKosten;
498                    besteEntscheidung = aktuellerFall;
499                }
500            }
501        }
502
503        if (kostenBesteEntscheidung == int.MaxValue)
504        {
505            // TO-DO: Das darf nicht passieren!
506            AlgoEntferneDebugNachricht(
507                "FEHLER: Der Algorithmus hat keine Loesung gefunden! Das darf ↵
508                nicht passieren!", 0);
509        }
509    else

```

```

F:\Users\meister\Documents\Visual Studio ...\\RestaurantAlgorithmen\Algorithmen.cs 9
510         {
511             AlgoEntferneDebugNachricht(
512                 "Der Algorithmus hat die Loesung " + besteEntscheidung.ToString() ↵
513                 + " gefunden.\n\n", 0);
514             AlgoEntferneDebugErgebnis("Originalkosten sind " +
515                 AlgoEntferneBoolArrayDebugNachricht
516                 (tischOriginalKosten) + ".");
517             AlgoEntferneDebugErgebnis("Der Algorithmus hat die Loesung " +
518                 besteEntscheidung.ToString() +
519                     " gefunden.\n\n");
520         }
521     }
522     /// <summary>
523     /// Algorithmus zum Entfernen von Gruppen, erwartet eine Tisch-Klasse und gibt ↵
524     /// eine Liste mit zu entfernenden Gruppen zurueck.
525     public static List<IGruppe> EinfachesRestaurant_EntferneGruppen(Tisch tisch, ↵
526     int maximaleIntervallGroesse)
527     {
528         AlgoEntferneDebugNachricht("Algorithmus auf "
529             + tisch.ToString() + " mit maximaler
530             Intervallgroesse " +
531             maximaleIntervallGroesse.ToString() + " ↵
532             gestartet.", 0);
533
534         // Tisch-Klasse in Tisch-Kosten-Struktur umwandeln
535         EinfachesRestaurant_TischKosten kosten =
536             EinfachesRestaurant_BerechneKosten(tisch, maximaleIntervallGroesse);
537
538         // Algorithmus starten
539         EinfachesRestaurant_TischKosten besteEntscheidung =
540             EinfachesRestaurant_EntferneGruppen(kosten,
541             maximaleIntervallGroesse);
542
543         // Liste mit zu entfernenden Gruppen ausgeben
544         List<IGruppe> rueckgabe = new List<IGruppe>();
545         HashSet<IGruppe> hashGruppen = new HashSet<IGruppe>();
546
547         // Jede Gruppe nur einmal in die Liste hinzufuegen, deshalb eine ↵
548         // Hashtabelle verwenden
549         for (int i = 0; i < besteEntscheidung.Plaetze.Length; i++)
550         {
551             if (kosten.Plaetze[i] != besteEntscheidung.Plaetze[i] &&
552             besteEntscheidung.Plaetze[i] == 0)
553             {
554                 hashGruppen.Add(tisch.GetSitzplaetze(i));
555             }
556
557         }
558
559         foreach (IGruppe aktuelleGruppe in hashGruppen)
560         {
561             rueckgabe.Add(aktuelleGruppe);
562         }
563
564         return rueckgabe;
565     }
566
567     /// <summary>
568     /// Rekursiver Teil des EntferneGruppen-Algorithmus.
569     /// </summary>
570     /// <param name="kosten">Tisch-Kosten-Struktur, die mit jedem
571     /// Rekursionsschritt kleiner wird (Divide-and-Conquer)</param>
572     /// <param name="original_kosten">Boolsches Array, das auf wahr steht, wenn
573     /// eine Luecke schon vor Anwendung des Algorithmus zu gross war</param>
574     /// <param name="rekursionstiefe">Aktuelle Rekursionstiefe (z.Zt. nur fuer
575     /// Debug-Zwecke und nicht etwa als Abbruchbedingung)</param>
576     /// <returns>Tisch-Kosten-Struktur</returns>

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 10
566     private static EinfachesRestaurant_TischKosten
567         EinfachesRestaurant_EntferneGruppen(EinfachesRestaurant_TischKosten kosten, bool[] ↵
568             original_kosten, int rekursionstiefe)
569         {
570             AlgoEntferneDebugNachricht("/*R* Rekursiver Aufruf gestartet mit Kosten " + ↵
571             kosten.ToString() + ".", rekursionstiefe);
572
573             // Abbruchbedingung: Keine entfernbarer Gruppe am Tisch
574             bool gruppeGefunden = false;
575
576             for (int i = 0; i < kosten.Plaetze.Length; i++)
577             {
578                 if (kosten.Plaetze[i] > 0)
579                 {
580                     gruppeGefunden = true;
581                     break;
582                 }
583             }
584             if (!gruppeGefunden)
585             {
586                 AlgoEntferneDebugNachricht("/*R* Es gibt keine entfernbarer Gruppe.", ↵
rekursionstiefe);
587                 return kosten;
588             }
589             // Divide and Conquer
590
591             // Finde eine Stelle zum Teilen
592             int mitte = kosten.Plaetze.Length/2;
593             int aktuellePosition = mitte;
594
595             while (aktuellePosition < kosten.Plaetze.Length && kosten.Plaetze[mitte] = ↵
= kosten.Plaetze[aktuellePosition])
596             {
597                 aktuellePosition++;
598             }
599
600             if (aktuellePosition == kosten.Plaetze.Length)
601             {
602                 // Es wurde keine Stelle zur Trennung gefunden, weil der rechte Teil ↵
aus einer Gruppe besteht
603                 aktuellePosition = mitte;
604
605                 while (aktuellePosition > -1 && kosten.Plaetze[mitte] == kosten. ↵
Plaetze[aktuellePosition])
606                 {
607                     aktuellePosition--;
608                 }
609
610                 if (aktuellePosition == -1)
611                 {
612                     // Das ganze Array besteht aus einer Gruppe, Abbruchbedingung
613                     // TO-DO: WIRKLICH ABBRUCHBEDINGUNG??
614                     return kosten;
615                 }
616
617                 aktuellePosition++;
618             }
619
620             // Bei aktuellePosition beginnt eine neue Gruppe
621             mitte = aktuellePosition;
622
623             AlgoEntferneDebugNachricht("/*R* Teilen des Arrays bei " + mitte.ToString() ↵
+ ", linker Teil [0; " + mitte.ToString() + "[.", rekursionstiefe);
624
625             // Generiere alle 4 Entscheidungs möglichkeiten
626             EinfachesRestaurant_TischKosten besteEntscheidung = new ↵
EinfachesRestaurant_TischKosten();
627             int kostenBesteEntscheidung = int.MaxValue; // Eigentlich unendlich

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 11

627
628         for (int i = 1; i < 5; i++)
629     {
630         EinfachesRestaurant_TischKosten linkerTeil, rechterTeil;
631         bool[] original_kosten_linkerTeil, original_kosten_rechterTeil;
632
633         linkerTeil = new EinfachesRestaurant_TischKosten();
634         rechterTeil = new EinfachesRestaurant_TischKosten();
635
636         linkerTeil.MaximaleLueckenGroesse = rechterTeil.MaximaleLueckenGroesse ↵
637         = kosten.MaximaleLueckenGroesse;
638         linkerTeil.Plaetze = new int[mitte];
639         original_kosten_linkerTeil = new bool[mitte];
640
641         for (int j = 0; j < mitte; j++)
642     {
643             linkerTeil.Plaetze[j] = kosten.Plaetze[j];
644             original_kosten_linkerTeil[j] = original_kosten[j];
645         }
646
647         rechterTeil.Plaetze = new int[kosten.Plaetze.Length - mitte];
648         original_kosten_rechterTeil = new bool[kosten.Plaetze.Length - mitte];
649
650         for (int j = mitte; j < kosten.Plaetze.Length; j++)
651     {
652             rechterTeil.Plaetze[j - mitte] = kosten.Plaetze[j];
653             original_kosten_rechterTeil[j - mitte] = original_kosten[j];
654         }
655
656     {
657         _gruppenZaehler++;
658
659         // Entferne die Gruppe am rechten Rand im linkes Subarray oder ↵
660         markiere sie als fest
661         aktuellePosition = linkerTeil.Plaetze.Length - 1;
662         int aktuelleGruppe = linkerTeil.Plaetze[aktuellePosition];
663
664         while (aktuellePosition > -1)
665     {
666             if (aktuelleGruppe == 0)
667             {
668                 // Der Arrayslot am Rand ist ein leerer Platz
669
670                 if (linkerTeil.Plaetze[aktuellePosition] > 0)
671                 {
672                     // Erste Gruppe gefunden, die auf ein leeres ↵
673                     Arrayintervall folgt
674                     aktuelleGruppe = linkerTeil.Plaetze[aktuellePosition];
675
676                     if (i == 1 || i == 2)
677                     {
678                         // Arrayslot einfach freigeben
679                         linkerTeil.Plaetze[aktuellePosition] = 0;
680                     }
681                     else
682                     {
683                         // Gruppe als fest markieren
684                         linkerTeil.Plaetze[aktuellePosition] = - ↵
685                         _gruppenZaehler;
686                     }
687                 }
688             }
689             else if (linkerTeil.Plaetze[aktuellePosition] < 0)
690             {
691                 // Ein Platz mit negativen Kosten darf nicht entfernt ↵
692                 werden!
693                 break;
694             }
695         }
696         else if (aktuelleGruppe != linkerTeil.Plaetze ↵

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 12
    [aktuellePosition])
    {
        // Jetzt beginnt eine neue Gruppe
        break;
    }
    else if (aktuelleGruppe < 0)
    {
        // Eine Gruppe mit negativen Kosten darf nicht entfernt ↵
werden!
        break;
    }
    else
    {
        if (i == 1 || i == 2)
        {
            // Arrayslot einfach freigeben
            linkerTeil.Plaetze[aktuellePosition] = 0;
        }
        else
        {
            // Gruppe als fest markieren
            linkerTeil.Plaetze[aktuellePosition] = - ↵
_gruppenZaehler;
        }
    }
    aktuellePosition--;
}
}

{
    _gruppenZaehler++;

// Entferne die Gruppe am linken Rand im rechten Subarray oder ↵
markiere sie als fest
    aktuellePosition = 0;
    int aktuelleGruppe = rechterTeil.Plaetze[0];

    while (aktuellePosition < rechterTeil.Plaetze.Length)
    {
        if (aktuelleGruppe == 0)
        {
            // Der Arrayslot am Rand ist ein leerer Platz
            if (rechterTeil.Plaetze[aktuellePosition] > 0)
            {
                // Erste Gruppe gefunden, die auf ein leeres ↵
Arrayintervall folgt
                aktuelleGruppe = rechterTeil.Plaetze[aktuellePosition] ↵
;

                if (i == 1 || i == 3)
                {
                    // Arrayslot einfach freigeben
                    rechterTeil.Plaetze[aktuellePosition] = 0;
                }
                else
                {
                    // Gruppe als fest markieren
                    rechterTeil.Plaetze[aktuellePosition] = - ↵
_gruppenZaehler;
                }
            }
            else if (rechterTeil.Plaetze[aktuellePosition] < 0)
            {
                // Ein Platz mit negativen Kosten darf nicht entfernt ↵
werden!
                break;
            }
        }
    }
}

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 13
    }
    else if (aktuelleGruppe != rechterTeil.Plaetze
    [aktuellePosition])
    {
        // Jetzt beginnt eine neue Gruppe
        break;
    }
    else if (aktuelleGruppe < 0)
    {
        // Eine Gruppe mit negativen Kosten darf nicht entfernt
        werden!
        break;
    }
    else
    {
        if (i == 1 || i == 3)
        {
            // Arrayslot einfach freigeben
            rechterTeil.Plaetze[aktuellePosition] = 0;
        }
        else
        {
            // Gruppe als fest markieren
            rechterTeil.Plaetze[aktuellePosition] = -
            _gruppenZaehler;
        }
    }
    aktuellePosition++;
}
}

AlgoEntferneDebugNachricht(" *R* Aktueller Fall " + i.ToString() + "
mit linker Teil " +
" + rechterTeil.ToString() + ".",
rekursionstiefe);

EinfachesRestaurant_TischKosten vereinigteTischArrays =
    EinfachesRestaurant_VereinigeTischeArrays(linkerTeil, rechterTeil);
;

if (EinfachesRestaurant_PruefeTischStrukturOhneUmlauf
(vereinigteTischArrays, original_kosten))
{
    // Mache nur dann weiter, wenn kein offensichtlicher Fehler
    begangen wurden: MaximaleLuckenGroesse muss eingehalten werden!
    AlgoEntferneDebugNachricht(
        "*R* Das vereinigte Tischarray hat den ersten Trivialtest
    ueberstanden.", rekursionstiefe);
    EinfachesRestaurant_TischKosten linkerTeilBearbeitet =
        EinfachesRestaurant_EntferneGruppen(linkerTeil,
    original_kosten_linkerTeil, rekursionstiefe + 1);
    EinfachesRestaurant_TischKosten rechterTeilBearbeitet =
        EinfachesRestaurant_EntferneGruppen(rechterTeil,
    original_kosten_rechterTeil, rekursionstiefe + 1);
    EinfachesRestaurant_TischKosten rekursivesErgebnis =
        EinfachesRestaurant_VereinigeTischeArrays(linkerTeilBearbeitet
    , rechterTeilBearbeitet);

    if (rekursivesErgebnis.Kosten != int.MaxValue)
    {
        // Diese Loesung ist auf den ersten Blick gueltig

        if (EinfachesRestaurant_PruefeTischStrukturOhneUmlauf
(rekursivesErgebnis, original_kosten))
        {
            AlgoEntferneDebugNachricht(
                "*R* Vereinigtes Tischarray " + vereinigteTischArrays.
ToString() +

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 14
810                                     " nach rekursivem Aufruf " +
811                                     rekursivesErgebnis.ToString() + " ist gueltig mit " ↵
812                                     Kosten " +
813                                     rekursivesErgebnis.Kosten.ToString() + " und " ↵
814                                     AlgoEntferneBoolArrayDebugNachricht(original_kosten) + ↵
815                                     "." , rekursionstiefe);
816                                     if (rekursivesErgebnis.Kosten < kostenBesteEntscheidung)
817                                     {
818                                         kostenBesteEntscheidung = rekursivesErgebnis.Kosten;
819                                         besteEntscheidung = rekursivesErgebnis;
820                                     }
821                                     else
822                                     {
823                                         AlgoEntferneDebugNachricht(
824                                         "*R* Vereinigtes Tischarray " + vereinigteTischArrays. ↵
825                                         ToString() +
826                                         " nach rekursivem Aufruf " +
827                                         rekursivesErgebnis.ToString() + " ist UNGueltig mit " ↵
828                                         Kosten " +
829                                         rekursivesErgebnis.Kosten.ToString() + " und " ↵
830                                         Originalkosten " +
831                                         AlgoEntferneBoolArrayDebugNachricht(original_kosten) + ↵
832                                         "." , rekursionstiefe);
833                                         AlgoEntferneDebugNachricht(
834                                         "*R* Rekursiver Aufruf hat ein ungueltiges Ergebnis " ↵
835                                         erzeugt! *ERR?*.", rekursionstiefe);
836                                     }
837                                     else
838                                     {
839                                         AlgoEntferneDebugNachricht("*R* Der aktuelle Fall wurde schon " ↵
840                                         vorzeitig als ungueltig erkannt.", rekursionstiefe);
841                                     }
842                                     if (kostenBesteEntscheidung == int.MaxValue)
843                                     {
844                                         AlgoEntferneDebugNachricht("*R* Keiner der 4 Faelle ist gueltig!", rekursionstiefe);
845                                         besteEntscheidung.Kosten = int.MaxValue;
846                                     }
847                                     return besteEntscheidung;
848                                 }
849                                 /// <summary>
850                                 /// Gibt ein Array zurueck, das genau dann auf "wahr" steht, wenn ein freies Platzintervall zu gross ist.
851                                 /// </summary>
852                                 /// <param name="kosten">Kosten-Tischstruktur</param>
853                                 /// <returns>Array</returns>
854                                 private static bool[] EinfachesRestaurant_BerechneOriginalKosten
855                                 (EinfachesRestaurant_TischKosten kosten)
856                                 {
857                                     bool[] tmp = new bool[kosten.Plaetze.Length];
858                                     // Array mit "falsch" initialisieren
859                                     for (int i = 0; i < kosten.Plaetze.Length; i++)
860                                     {
861                                         tmp[i] = false;
862                                     }
863                                     if (kosten.MaximaleLueckenGroesse >= kosten.Plaetze.Length)

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 15
868         {
869             return tmp;
870         }
871
872         int startAktuellerLauf = -1;
873
874         for (int i = 0; i < 2*kosten.Plaetze.Length; i++)
875         {
876             if (kosten.Plaetze[i%kosten.Plaetze.Length] == 0)
877             {
878                 // Platz ist nicht besetzt
879                 if (startAktuellerLauf == -1)
880                 {
881                     startAktuellerLauf = i;
882                 }
883             }
884         else
885         {
886             if (startAktuellerLauf != -1)
887             {
888                 // Dieser Platz ist nicht besetzt und es hat schon ein Lauf
889                 begonnen
890                 if (i - startAktuellerLauf > kosten.MaximaleLueckenGroesse)
891                 {
892                     // Array hier auf wahr stellen
893                     for (int j = startAktuellerLauf; j < i; j++)
894                     {
895                         tmp[j%kosten.Plaetze.Length] = true;
896                     }
897
898                     startAktuellerLauf = -1;
899                 }
900             }
901         }
902
903         if (startAktuellerLauf == 0)
904         {
905             // Bei der Position 0 hat ein Lauf begonnen, der nie endete -> Ganzes
906             Array muss auf "wahrt" stehen
907             for (int j = 0; j < kosten.Plaetze.Length; j++)
908             {
909                 tmp[j] = true;
910             }
911
912         return tmp;
913     }
914
915     public static bool EinfachesRestaurant_PruefeTischStrukturOhneUmlauf
916     (EinfachesRestaurant_TischKosten kosten, bool[] kosten_original)
917     {
918         int arrayGroesse = kosten.Plaetze.Length;
919         int aktuelleIntervallGroesse = 0;
920
921         if (kosten.MaximaleLueckenGroesse >= arrayGroesse)
922         {
923             return true;
924         }
925
926         aktuelleIntervallGroesse = 0;
927
928         for (int i = 0; i < arrayGroesse; i++)
929         {
930             if (kosten.Plaetze[i] == 0)
931             {
932                 // Der aktuelle Platz ist nicht besetzt
933                 if (kosten_original[i])
934                 {
935                     // Dieses Intervall ignorieren

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 16

935             if (aktuelleIntervallGroesse > 0)
936             {
937                 // Das bereits zu grosse Intervall wurde vergroessert
938                 return false;
939             }
940
941             aktuelleIntervallGroesse = 0;
942         }
943     }
944
945     // Intervall nicht ignorieren
946     aktuelleIntervallGroesse++;
947 }
948 }
949 else
950 {
951     // Der aktuelle Platz ist besetzt, jetzt Intervallgroesse pruefen
952     if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
953     {
954         // Intervall ist zu gross
955         return false;
956     }
957
958     aktuelleIntervallGroesse = 0;
959 }
960 }
961
962 if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
963 {
964     return false;
965 }
966
967 // Von der anderen Richtung durchsuchen
968 aktuelleIntervallGroesse = 0;
969
970 for (int i = arrayGroesse - 1; i > -1; i--)
971 {
972     if (kosten.Plaetze[i] == 0)
973     {
974         // Der aktuelle Platz ist nicht besetzt
975         if (kosten_original[i])
976         {
977             // Dieses Intervall ignorieren
978             if (aktuelleIntervallGroesse > 0)
979             {
980                 // Das bereits zu grosse Intervall wurde vergroessert
981                 return false;
982             }
983
984             aktuelleIntervallGroesse = 0;
985         }
986     else
987     {
988         // Intervall nicht ignorieren
989         aktuelleIntervallGroesse++;
990     }
991 }
992 else
993 {
994     // Der aktuelle Platz ist besetzt, jetzt Intervallgroesse pruefen
995     if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
996     {
997         // Intervall ist zu gross
998         return false;
999     }
1000
1001     aktuelleIntervallGroesse = 0;
1002 }
1003 }
1004

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 17
1005         if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
1006         {
1007             return false;
1008         }
1009     }
1010 }
1011 }
1012
1013     private static bool EinfachesRestaurant_PruefeTischStruktur
1014     (EinfachesRestaurant_TischKosten kosten, bool[] kosten_original)
1015     {
1016         bool aktuelleIgnorieren = kosten_original[0]; // Steht auf "wahr", wenn
1017         das aktuelle Intervall ignoriert werden soll
1018         int arrayGroesse = kosten.Plaetze.Length; // Groesse des Arrays (Slots)
1019         int aktuelleIntervallGroesse = 0; // Groesse des aktuellen
1020         Intervalls, bis zum aktuellen Index i, also so weit wie bisher "erforscht"
1021
1022         if (kosten.MaximaleLueckenGroesse >= arrayGroesse)
1023         {
1024             return true;
1025         }
1026
1027         aktuelleIntervallGroesse = 0;
1028
1029         for (int i = 0; i < 2 * arrayGroesse; i++)
1030         {
1031             if (kosten.Plaetze[i % arrayGroesse] == 0)
1032             {
1033                 // Der aktuelle Platz ist nicht besetzt
1034                 if (kosten_original[i % arrayGroesse])
1035                 {
1036                     // Dieses Intervall ignorieren
1037                     if (aktuelleIntervallGroesse > 0)
1038                     {
1039                         // Das bereits zu grosse Intervall wurde vergroessert
1040                         return false;
1041                     }
1042                 }
1043             }
1044             else
1045             {
1046                 // Intervall nicht ignorieren
1047                 aktuelleIntervallGroesse++;
1048             }
1049         }
1050         else
1051         {
1052             // Der aktuelle Platz ist besetzt, jetzt Intervallgroesse pruefen
1053             if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
1054             {
1055                 // Intervall ist zu gross
1056                 return false;
1057             }
1058             aktuelleIntervallGroesse = 0;
1059         }
1060
1061         // Von der anderen Seite durchsuchen
1062         aktuelleIntervallGroesse = 0;
1063
1064         for (int i = 2 * arrayGroesse - 1; i > -1; i--)
1065         {
1066             if (kosten.Plaetze[i % arrayGroesse] == 0)
1067             {
1068                 // Der aktuelle Platz ist nicht besetzt
1069                 if (kosten_original[i % arrayGroesse])
1070                 {
1071                     // Dieses Intervall ignorieren

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 18
1072             if (aktuelleIntervallGroesse > 0)
1073             {
1074                 // Das bereits zu grosse Intervall wurde vergroessert
1075                 return false;
1076             }
1077
1078             aktuelleIntervallGroesse = 0;
1079         }
1080     else
1081     {
1082         // Intervall nicht ignorieren
1083         aktuelleIntervallGroesse++;
1084     }
1085 }
1086 else
1087 {
1088     // Der aktuelle Platz ist besetzt, jetzt Intervallgroesse pruefen
1089     if (aktuelleIntervallGroesse > kosten.MaximaleLueckenGroesse)
1090     {
1091         // Intervall ist zu gross
1092         return false;
1093     }
1094
1095     aktuelleIntervallGroesse = 0;
1096 }
1097 }
1098
1099     return true;
1100 }
1101
1102     private static EinfachesRestaurant_TischKosten
EinfachesRestaurant_VereinigeTischeArrays(EinfachesRestaurant_TischKosten kosten1,
EinfachesRestaurant_TischKosten kosten2)
1103 {
1104     EinfachesRestaurant_TischKosten tmp = new EinfachesRestaurant_TischKosten
();
1105     tmp.Plaetze = new int[kosten1.Plaetze.Length + kosten2.Plaetze.Length];
1106     tmp.MaximaleLueckenGroesse = kosten1.MaximaleLueckenGroesse;
1107
1108     for (int i = 0; i < kosten1.Plaetze.Length; i++)
1109     {
1110         tmp.Plaetze[i] = kosten1.Plaetze[i];
1111     }
1112
1113     for (int i = 0; i < kosten2.Plaetze.Length; i++)
1114     {
1115         tmp.Plaetze[i + kosten1.Plaetze.Length] = kosten2.Plaetze[i];
1116     }
1117
1118     tmp.Kosten = EinfachesRestaurant_BerechneKosten(tmp);
1119     return tmp;
1120 }
1121
1122     private static int EinfachesRestaurant_BerechneKosten
(EinfachesRestaurant_TischKosten kosten)
1123 {
1124     int tmp_kosten = 0;
1125
1126     for (int i = 0; i < kosten.Plaetze.Length; i++)
1127     {
1128         if (kosten.Plaetze[i] != 0)
1129         {
1130             tmp_kosten++;
1131         }
1132
1133         // EDIT: Es gibt jetzt keine individuellen Kosten fuer jede Gruppe
mehr.
1134         // tmp_kosten += Math.Abs(kosten.Plaetze[i]);
1135     }
1136

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 19
1137         return tmp_kosten;
1138     }
1139
1140     public static EinfachesRestaurant_TischKosten
1141         EinfachesRestaurant_KopiereStruktur(EinfachesRestaurant_TischKosten kosten)
1142     {
1143         EinfachesRestaurant_TischKosten tmp;
1144         tmp.Kosten = kosten.Kosten;
1145         tmp.MaximaleLueckenGroesse = kosten.MaximaleLueckenGroesse;
1146         tmp.Plaetze = new int[kosten.Plaetze.Length];
1147
1148         for (int i = 0; i < kosten.Plaetze.Length; i++)
1149         {
1150             tmp.Plaetze[i] = kosten.Plaetze[i];
1151         }
1152
1153         return tmp;
1154     }
1155
1156
1157     public struct EinfachesRestaurant_TischKosten
1158     {
1159         public int[] Plaetze;
1160         public int Kosten;
1161         public int MaximaleLueckenGroesse;
1162
1163         /// <summary>
1164         /// Wandelt die Struktur in einen String um, damit dieser auf der Konsole
1165         /// ausgegeben werden kann.
1166         /// </summary>
1167         public override string ToString()
1168     {
1169         string rueckgabe = "{ ";
1170
1171         for (int i = 0; i < Plaetze.Length; i++)
1172         {
1173             rueckgabe += Plaetze[i].ToString();
1174
1175             if (i < Plaetze.Length - 1) rueckgabe += " ; ";
1176
1177         rueckgabe += "}";
1178         return rueckgabe;
1179     }
1180
1181         /// <summary>
1182         /// Gibt "wahr" zurueck, falls ein Platz frei ist.
1183         /// </summary>
1184         /// <param name="index">Index des Platzes</param>
1185         /// <returns>Wahrheitswert</returns>
1186         public bool IstPlatzFrei(int index)
1187     {
1188         if (Plaetze[index] == 0)
1189         {
1190             return true;
1191         }
1192         else
1193         {
1194             return false;
1195         }
1196     }
1197 }
1198
1199 #region Prioritaetswarteschlange, Heap
1200 [Serializable]
1201 public class Heap<T> where T : IComparable
1202 {
1203     /// <summary>

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 20
1205     /// Ein dynamisch vergroesserbares Array.
1206     /// </summary>
1207     private ArrayList _heap;
1208
1209     /// <summary>
1210     /// Der groesste vorkommende Index im Array (Anzahl der Elemente - 1).
1211     /// </summary>
1212     private int _laenge;
1213
1214     public Heap()
1215     {
1216         _heap = new ArrayList(0);
1217         _laenge = -1;
1218     }
1219
1220     /// <summary>
1221     /// Gibt die Groesse des Heaps (Anzahl der Elemente) zurueck.
1222     /// </summary>
1223     public int Groesse()
1224     {
1225         return _laenge + 1;
1226     }
1227
1228     /// <summary>
1229     /// Fuegt einen Datensatz im Heap ein und ruft Upheap auf.
1230     /// </summary>
1231     /// <param name="datensatz">Datensatz</param>
1232     public void Einfuegen(T datensatz)
1233     {
1234         _laenge++;
1235
1236         if (_heap.Count - 1 - _laenge < 0)
1237         {
1238             // Einen neuen Slot erstellen
1239             _heap.Add(0);
1240         }
1241
1242         _heap[_laenge] = datensatz;
1243         Upheap(_laenge);
1244     }
1245
1246     /// <summary>
1247     /// Entfernt das Maximum aus dem Heap und gibt dieses zurueck.
1248     /// </summary>
1249     /// <returns>Datensatz mit dem maximalen Schluessel</returns>
1250     public T EntferneMaximum()
1251     {
1252         var maximum = (T)_heap[0];
1253
1254         // Verschiebe den letzten Array-Eintrag nach oben
1255         _heap[0] = _heap[_laenge];
1256         _heap.RemoveAt(_laenge);
1257
1258         _laenge--;
1259         if (_laenge > -1) Downheap(0);
1260
1261         return maximum;
1262     }
1263
1264     /// <summary>
1265     /// Vertauscht einen Datensatz so lange mit seinem Elternknoten, bis er an der rechten Stelle ist, also kein kleinerer Elternknoten mehr vorhanden ist.
1266     /// </summary>
1267     /// <param name="index">Index des Knotens (im Array)</param>
1268     private void Upheap(int index)
1269     {
1270         // Zu verschiebenden Knoten zwischenspeichern
1271         var wert = (T)_heap[index];
1272         int aktuellerKnoten = index;
1273

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 21
1274          // Knoten so lange nach oben schieben, bis er keinen kleineren ↵
1275          Elternknoten mehr hat (Maximum oben)
1276          while (((T)_heap[aktuellerKnoten / 2]).CompareTo(wert) < 0 &&
1277          aktuellerKnoten > 0)
1278          {
1279              // Der Elternknoten hat den Index floor(aktuellerKnoten / 2), da es ↵
1280              // sich um einen vollstaendigen Binaerbaum handelt
1281              _heap[aktuellerKnoten] = _heap[aktuellerKnoten / 2];
1282              aktuellerKnoten /= 2;
1283          }
1284
1285      _heap[aktuellerKnoten] = wert;
1286  }
1287
1288  /// <summary>
1289  /// Schiebt einen Knoten so lange nach unten, bis er keinen groesseren ↵
1290  /// Nachfolger (Kinderknoten) mehr hat.
1291  /// </summary>
1292  /// <param name="index">Index des Knotens</param>
1293  private void Downheap(int index)
1294  {
1295      int aktuellerKnoten = index;
1296      var wert = (T)_heap[index];
1297
1298      while (true)
1299      {
1300          bool istLinkerGroesser = false;
1301          bool istRechterGroesser = false;
1302
1303          if (2 * aktuellerKnoten < _laenge + 1)
1304          {
1305              if (((T)_heap[2 * aktuellerKnoten]).CompareTo(wert) > 0)
1306              {
1307                  istLinkerGroesser = true;
1308              }
1309
1310              if (2 * aktuellerKnoten + 1 < _laenge + 1)
1311              {
1312                  if (((T)_heap[2 * aktuellerKnoten + 1]).CompareTo(wert) > 0)
1313                  {
1314                      istRechterGroesser = true;
1315                  }
1316              }
1317          }
1318
1319          else
1320          {
1321              // Es gibt keinen rechten Nachfolger
1322          }
1323
1324          else
1325          {
1326              // Es gibt keinen linken Nachfolger und damit auch keinen rechten ↵
1327              Nachfolger, weil der Baum von links nach rechts aufgebaut wird
1328          }
1329
1330          if (istLinkerGroesser && istRechterGroesser)
1331          {
1332              if (((T)_heap[2 * aktuellerKnoten]).CompareTo((T)_heap[2 *
aktuellerKnoten + 1]) > 0)
1333              {
1334                  // Der linke Knoten ist groesser als der rechte Knoten
1335                  istRechterGroesser = false;
1336              }
1337          }
1338
1339          if (istLinkerGroesser)
1340          {

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 22

1338         // Vertausche mit dem linken Nachfolger
1339         _heap[aktuellerKnoten] = _heap[2 * aktuellerKnoten];
1340         aktuellerKnoten = 2 * aktuellerKnoten;
1341     }
1342     else if (istRechterGroesser)
1343     {
1344         // Vertausche mit dem rechten Nachfolger
1345         _heap[aktuellerKnoten] = _heap[2 * aktuellerKnoten + 1];
1346         aktuellerKnoten = 2 * aktuellerKnoten + 1;
1347     }
1348     else
1349     {
1350         // Es gibt keine groesseren Kinderknoten mehr
1351         _heap[aktuellerKnoten] = wert;
1352         return;
1353     }
1354 }
1355 }
1356 #endregion
1358 #region Dequeue (Double ended queue)
1359 [Serializable]
1360 public class Dequeue<T>
1361 {
1363     /// <summary>
1364     /// Erstes Element in der Dequeue
1365     /// </summary>
1366     private DequeueItem<T> _anfang;
1367
1368     /// <summary>
1369     /// Letztes Element in der Dequeue
1370     /// </summary>
1371     private DequeueItem<T> _ende;
1372
1373     private int _groesse;
1374
1375     public Dequeue()
1376     {
1377         _anfang = default(DequeueItem<T>);
1378         _ende = default(DequeueItem<T>);
1379         _groesse = 0;
1380     }
1381
1382     public void EinfuegenAnfang(T objekt)
1383     {
1384         DequeueItem<T> neuesElement = new DequeueItem<T>();
1385         neuesElement.Objekt = objekt;
1386
1387         if (_anfang == default(DequeueItem<T>))
1388         {
1389             // Es gibt noch keinen Anfang, die Dequeue ist leer
1390             _anfang = neuesElement;
1391             _ende = neuesElement;
1392         }
1393         else
1394         {
1395             // Die Dequeue ist nicht leer
1396             DequeueItem<T> alterAnfang = _anfang;
1397             neuesElement.Naechstes = alterAnfang;
1398             _anfang = neuesElement;
1399             alterAnfang.Vorheriges = neuesElement;
1400         }
1401
1402         _groesse++;
1403     }
1404
1405     public void EinfuegenEnde(T objekt)
1406     {
1407         DequeueItem<T> neuesElement = new DequeueItem<T>();

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 23
1408         neuesElement.Objekt = objekt;
1409
1410         if (_ende == default(DequeueItem<T>))
1411     {
1412             // Es gibt noch keinen Anfang, die Dequeue ist leer
1413             _anfang = neuesElement;
1414             _ende = neuesElement;
1415         }
1416     else
1417     {
1418         // Die Dequeue ist nicht leer
1419         DequeueItem<T> altesEnde = _ende;
1420         neuesElement.Vorheriges = altesEnde;
1421         _ende = neuesElement;
1422         altesEnde.Naechstes = neuesElement;
1423     }
1424
1425     _groesse++;
1426 }
1427
1428     public T EntferneAnfang()
1429     {
1430         T rueckgabe = _anfang.Objekt;
1431
1432         if (_groesse == 1)
1433     {
1434             // Es wird das letzte Element entfernt
1435             _anfang = default(DequeueItem<T>);
1436             _ende = default(DequeueItem<T>);
1437         }
1438     else
1439     {
1440         DequeueItem<T> neuerAnfang = _anfang.Naechstes;
1441         _anfang = neuerAnfang;
1442         _anfang.Vorheriges = default(DequeueItem<T>);
1443     }
1444
1445     if (_groesse == 2)
1446     {
1447         // Es muss auch der Zeiger von _ende veraendert werden
1448         _ende = _anfang;
1449     }
1450
1451     _groesse--;
1452     return rueckgabe;
1453 }
1454
1455     public T EntferneEnde()
1456     {
1457         T rueckgabe = _ende.Objekt;
1458
1459         if (_groesse == 1)
1460     {
1461             // Es wird das letzte Element entfernt
1462             _anfang = default(DequeueItem<T>);
1463             _ende = default(DequeueItem<T>);
1464         }
1465     else
1466     {
1467         DequeueItem<T> neuesEnde = _ende.Vorheriges;
1468         _ende = neuesEnde;
1469         _ende.Naechstes = default(DequeueItem<T>);
1470     }
1471
1472     if (_groesse == 2)
1473     {
1474         // Es muss auch der Zeiger von _anfang veraendert werden
1475         _anfang = _ende;
1476     }
1477

```

F:\Users\meister\Documents\Visual Studio...\\RestaurantAlgorithmen\Algorithmen.cs 24

```

1478         _groesse--;
1479         return rueckgabe;
1480     }
1481
1482     public int Groesse()
1483     {
1484         return _groesse;
1485     }
1486 }
1487
1488 [Serializable]
1489 class DequeueItem<T>
1490 {
1491     public DequeueItem<T> Naechstes;
1492     public DequeueItem<T> Vorheriges;
1493     public T Objekt;
1494
1495     public DequeueItem()
1496     {
1497         Naechstes = default(DequeueItem<T>);
1498         Vorheriges = default(DequeueItem<T>);
1499     }
1500 }
1501 #endregion
1502
1503 #region ObjectContainer Interface und implementierende Klassen
1504 public interface IObjectConatiner<T>
1505 {
1506     /// <summary>
1507     /// Entnimmt das erste Element aus dem Container. Dabei haengt es von der
1508     /// entsprechenden Implementation ab, welches Element das Erste ist.
1509     T HoleErstesElement();
1510
1511     /// <summary>
1512     /// Fuegt ein Element in den Container ein und ordnet es ggf. ein.
1513     /// </summary>
1514     void FuegeElementEin(T element);
1515
1516     /// <summary>
1517     /// Ermittelt die Anzahl der im Container enthaltenen Elemente.
1518     /// </summary>
1519     int Count();
1520
1521     /// <summary>
1522     /// Verringert die scheinbare Anzahl an Elementen im Container (Count) um
1523     /// einen konstanten Wert.
1524     int PersonenAusblenden
1525     {
1526         get;
1527         set;
1528     }
1529 }
1530
1531     /// <summary>
1532     /// Stellt eine generische Liste dar, die das IObjectContainer-Interface
1533     /// implementiert.
1534     /// </summary>
1535     /// <typeparam name="T">Beliebiger Typ</typeparam>
1536     [Serializable]
1537     public class ObjectList<T> : IObjectConatiner<T>
1538     {
1539         private List<T> _interneListe = new List<T>();
1540         private int _personenAusblenden = 0;
1541
1542         /// <summary>
1543         /// Entnimmt das erste Element aus der Liste.
1544         /// </summary>
1545         public T HoleErstesElement()

```

```

F:\Users\meister\Documents\Visual Studio...\RestaurantAlgorithmen\Algorithmen.cs 25

1545     {
1546         T rueckgabe = _interneListe[0];
1547         _interneListe.RemoveAt(0);
1548
1549         return rueckgabe;
1550     }
1551
1552     /// <summary>
1553     /// Fuegt ein Element am Ende der Liste ein.
1554     /// </summary>
1555     public void FuegeElementEin(T element)
1556     {
1557         _interneListe.Add(element);
1558     }
1559
1560     public int Count()
1561     {
1562         return _interneListe.Count - _personenAusblenden;
1563     }
1564
1565     public int PersonenAusblenden
1566     {
1567         get
1568         {
1569             return _personenAusblenden;
1570         }
1571         set
1572         {
1573             _personenAusblenden = value;
1574         }
1575     }
1576 }
1577
1578     /// <summary>
1579     /// Stellt einen generischen Heap dar, der das IObjectContainer-Interface
1580     /// implementiert.
1581     /// </summary>
1582     /// <typeparam name="T">Beliebiger Typ, der das IComparable-Interface
1583     /// implementiert</typeparam>
1584     [Serializable]
1585     public class ObjectHeap<T> : IObjectContainer<T> where T : IComparable
1586     {
1587         private Heap<T> _internerHeap = new Heap<T>();
1588         private int _personenAusblenden = 0;
1589
1590         /// <summary>
1591         /// Entnimmt das groesste Element aus dem Heap.
1592         /// </summary>
1593         public T HoleErstesElement()
1594         {
1595             return _internerHeap.EntferneMaximum();
1596         }
1597
1598         /// <summary>
1599         /// Fuegt ein Element in den Heap ein und setzt es an die richtige Stelle im
1600         /// Heap.
1601         /// </summary>
1602         public void FuegeElementEin(T element)
1603         {
1604             _internerHeap.Einfuegen(element);
1605         }
1606
1607         public int Count()
1608         {
1609             return _internerHeap.Groesse() - _personenAusblenden;
1610         }
1611
1612         public int PersonenAusblenden
1613         {
1614             get
1615         }

```

F:\Users\meister\Documents\Visual Studio...\\RestaurantAlgorithmen\Algorithmen.cs 26

```
1612     {
1613         return _personenAusblenden;
1614     }
1615     set
1616     {
1617         _personenAusblenden = value;
1618     }
1619 }
1620 }
1621 #endregion
1622 }
1623 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\OberGaesteAuswahl.cs 1
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen
7 {
8     class OberGaesteAuswahl
9     {
10         private List<Gast> _potentielleGaeste = new List<Gast>();
11
12         /// <summary>
13         /// Speichert den Profit (zu maximieren) der mit der besten Entscheidung
14         /// erreicht werden kann.
15         /// </summary>
16         private int[,] _profitMatrix;
17
18         /// <summary>
19         /// Speichert, welche Entscheidung zur besten Loesung gefuehrt hat, damit die Einzelentscheidungen ermittelt werden koennen.
20         /// </summary>
21         private EntscheidungMatrix[,] _pfadMatrix;
22
23         public int AnzahlDerSitzplaetze = 10;
24
25         public OberGaesteAuswahl()
26         {
27             // Dummy hinzufuegen, damit die Indizes spaeter passen
28             _potentielleGaeste.Add(new Gast(0, 0));
29         }
30
31         public void GastHinzufuegen(int sitzplaetze, int profit)
32         {
33             _potentielleGaeste.Add(new Gast(sitzplaetze, profit));
34         }
35
36         public void GastHinzufuegen(Gast gast)
37         {
38             _potentielleGaeste.Add(gast);
39         }
40
41         public List<Gast> ErmittleBesteAuswahl()
42         {
43             // Dimensioniere Profit-Matrix
44             _profitMatrix = new int[AnzahlDerSitzplaetze + 1, _potentielleGaeste.Count + 1];
45             _pfadMatrix = new EntscheidungMatrix[AnzahlDerSitzplaetze + 1,
46             _potentielleGaeste.Count + 1];
47
48             // Generiere Basisfaelle
49             // In C# nicht notwendig, da Matrix auf standardmaessig auf Null steht und alle Basisfaelle den Profit Null haben
50             int aktuelleGruppe;
51             int restaurantGroesse;
52
53             for (aktuelleGruppe = 1; aktuelleGruppe < _potentielleGaeste.Count;
54                 aktuelleGruppe++) // Deshalb der Dummy
55             {
56                 for (restaurantGroesse = 1; restaurantGroesse <= AnzahlDerSitzplaetze;
57                     restaurantGroesse++)
58                 {
59                     if (_potentielleGaeste[aktuelleGruppe].Sitzplaetze <=
60                     restaurantGroesse)
61                     {
62                         // Wenn fuer die aktuelle Gruppe im Restaurant mit der
63                         // aktuellen Groesse genuegend Platz ist
64
65                         // Moeglichkeit 1: Der vorherige Fall mit der gleichen
66                         // Restaurantgroesse aber ohne die neue Gruppe
67                         int moeglichkeit1 = _profitMatrix[restaurantGroesse,
68
69

```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\OberGaesteAuswahl.cs 2
    aktuelleGruppe - 1];
61            int moeglichkeit2;
62
63            if (restaurantGroesse - _potentielleGaeste[aktuelleGruppe].
64 Sitzplaetze > -1)
65            {
66                // Moeglichkeit 2: Ein Restaurant der aktuellen Groesse
67                // minus der aktuellen Gruppengroesse, dann die neue Gruppe dazu
68                moeglichkeit2 =
69                    _profitMatrix[
70                        restaurantGroesse - _potentielleGaeste
71 [aktuelleGruppe].Sitzplaetze,
72                            aktuelleGruppe - 1] +
73                            _potentielleGaeste[aktuelleGruppe].Profit;
74            }
75            else
76            {
77                // Moeglichkeit 2 ist nicht moeglich
78                moeglichkeit2 = -1;
79            }
80            if (moeglichkeit1 > moeglichkeit2)
81            {
82                // Waehle den Fall, der profitabler ist
83
84                _profitMatrix[restaurantGroesse, aktuelleGruppe] =
85                _pfadMatrix[restaurantGroesse, aktuelleGruppe] = new
86                EntscheidungMatrix(restaurantGroesse, 1);
87            }
88            else
89            {
90                _profitMatrix[restaurantGroesse, aktuelleGruppe] =
91                _pfadMatrix[restaurantGroesse, aktuelleGruppe] = new
92                EntscheidungMatrix(restaurantGroesse -
93 _potentielleGaeste[
94 aktuelleGruppe].
95 Sitzplaetze, 2);
96            }
97            else
98            {
99                // Fuer diese Gruppe ist nicht genuegend Platz am Tisch, es
100 kommt also nur Moeglichkeit 1 in Frage
101                // Moeglichkeit 1: Der vorherige Fall mit der gleichen
102 Restaurantgroesse aber ohne die neue Gruppe
103                int moeglichkeit1 = _profitMatrix[restaurantGroesse,
104 aktuelleGruppe - 1];
105
106                _profitMatrix[restaurantGroesse, aktuelleGruppe] =
107                _pfadMatrix[restaurantGroesse, aktuelleGruppe] = new
108                EntscheidungMatrix(restaurantGroesse, 1);
109
110                Console.WriteLine("Matrix Eintrag P[" + restaurantGroesse.ToString() +
111 () + "][ " +
112 aktuelleGruppe.ToString() + "] = " +
113 _profitMatrix[restaurantGroesse, aktuelleGruppe].ToString() + ".");
114            }
115
116            // Ermittle die Einzelentscheidungen, die zur besten Loesung fuehren
117            List<Gast> rueckgabe = new List<Gast>();
118            aktuelleGruppe = _potentielleGaeste.Count - 1;
119            restaurantGroesse = AnzahlDerSitzplaetze;
120            Console.WriteLine("Profit am Schluss: " + _profitMatrix[restaurantGroesse,
121 aktuelleGruppe].ToString() + ".");

```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\OberGaesteAuswahl.cs 3
115
116         while (aktuelleGruppe != 0 && restaurantGroesse != 0)
117         {
118             EntscheidungMatrix entscheidung = _pfadMatrix[restaurantGroesse, ↵
119             aktuelleGruppe];
120
121             if (entscheidung.Entscheidung == 2)
122             {
123                 rueckgabe.Add(_potentielleGaeste[aktuelleGruppe]);
124             }
125
126             aktuelleGruppe--;
127             restaurantGroesse = entscheidung.MatrixSprungstelleSitzplaetze;
128         }
129
130     return rueckgabe;
131 }
132
133 /// <summary>
134 /// Eine Reservierung durch einen Gast, bzw. eine Gruppe.
135 /// </summary>
136 struct Gast
137 {
138     /// <summary>
139     /// Anzahl der benoetigten Sitzplaetze (Kosten).
140     /// </summary>
141     public int Sitzplaetze;
142
143     /// <summary>
144     /// Gesamtpreis fuer die Reservierung (Profit).
145     /// </summary>
146     public int Profit;
147
148     public Gast(int sitzplaetze, int profit)
149     {
150         Sitzplaetze = sitzplaetze;
151         Profit = profit;
152     }
153
154     public override string ToString()
155     {
156         return "Sitzplaetze: " + Sitzplaetze.ToString() + ", Profit: " + Profit. ↵
157         ToString();
158     }
159
160 /// <summary>
161 /// Speichert die beste Entscheidung (Moeglichkeit 1 oder 2) und den Wert ↵
162 /// "restaurantGroesse" (MatrixSprungstelleSitzplaetze), bei dem weitergemacht werden ↵
163 /// soll.
164 /// </summary>
165 struct EntscheidungMatrix
166 {
167     public int MatrixSprungstelleSitzplaetze; // Kann man sich eigentlich auch ↵
168     //sparen! Kann aus der Entscheidung berechnet werden.
169     public int Entscheidung;
170
171     public EntscheidungMatrix(int sprungstelle, int entscheidung)
172     {
173         MatrixSprungstelleSitzplaetze = sprungstelle;
174         Entscheidung = entscheidung;
175     }
176 }

```

```
F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\NormalePersonen.cs
1 using System;
2 using System.Collections.Generic;
3 using System.Drawing;
4 using System.Linq;
5 using System.Text;
6 using System.Windows.Forms;
7
8 namespace _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen
9 {
10     [Serializable]
11     public class NormalePersonen
12     {
13         public int anzahlDerPersonen;
14         public List<IGruppe> listeDerGruppen = new List<IGruppe>();
15         public Restaurant dasRestaurant;
16         private NormaleGruppe _letzteGruppe = default(NormaleGruppe);
17
18         public NormalePersonen(Restaurant restaurant)
19         {
20             dasRestaurant = restaurant;
21             restaurant.Tick += new TickEreignis(OnTick);
22         }
23
24         public NormalePersonen()
25         {
26             // Dummy fuer Serialisierung
27             anzahlDerPersonen = 0;
28         }
29
30         private void DebugNachrichtAnzeigen(string text)
31         {
32             StatischeObjekte.DebugNachricht("[Normale Personen] " + text);
33         }
34
35         public void OnTick(long uhrzeit)
36         {
37             DebugNachrichtAnzeigen("Tick-Ereignis bei Klasse NormalePersonen mit " +
anzahlDerPersonen.ToString() + " freien Personen.");
38
39             if (_letzteGruppe != default(NormaleGruppe) && !_letzteGruppe.
IstGruppePlatziert)
40             {
41                 // Die letzte Gruppe wurde noch nicht platziert
42                 DebugNachrichtAnzeigen("Letzte Gruppe wurde noch nicht platziert.");
43                 return;
44             }
45
46             // Entferne Gruppen
47             Random zufall = new Random();
48             int anzahlGruppen = zufall.Next(0, listeDerGruppen.Count);
49
50             // Entferne zufaellig anzahlGruppen Gruppen
51             for (int i = 0; i < anzahlGruppen; i++)
52             {
53                 // Waehle eine Gruppe zufaellig aus
54                 int index = zufall.Next(0, listeDerGruppen.Count);
55
56                 IGruppe gruppe = listeDerGruppen[index];
57
58                 // Wenn die Gruppe noch nicht platziert wurde darf sie nicht entfernt
werden
59                 if (gruppe.IstGruppePlatziert)
60                 {
61                     anzahlDerPersonen += gruppe.Personen.Count;
62                     gruppe.GruppeAufloesen();
63                     listeDerGruppen.RemoveAt(index);
64
65                     dasRestaurant.ChefOber.GruppeGeht(gruppe);
66                     DebugNachrichtAnzeigen("Gruppe " + i.ToString() + " (" + gruppe.
Personen.Count.ToString() + " Personen) gelöscht.");
67                 }
68             }
69         }
70     }
71 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\NormalePersonen.cs 2
67         }
68     }
69
70     // Erzeuge zufaellig neue Gruppen
71     // Wie viele Personen sollen verwendet werden?
72     int anzahlPersonen = zufall.Next(0, anzahlDerPersonen + 1);
73     DebugNachrichtAnzeigen("Es gibt " + anzahlDerPersonen.ToString() + " freie ↵
    Personen.");
74     DebugNachrichtAnzeigen("Es sollen " + anzahlPersonen.ToString() + " / " + ↵
anzahlDerPersonen.ToString() + " Personen verwendet werden.");
75     anzahlDerPersonen -= anzahlPersonen;
76
77     while (anzahlPersonen != 0)
78     {
79         // Erstelle eine neue Gruppe
80         // Wie viele Personen soll die Gruppe haben?
81         int gruppePersonen = zufall.Next(0, anzahlPersonen + 1);
82
83         if (gruppePersonen > 0)
84         {
85             anzahlPersonen -= gruppePersonen;
86
87             NormaleGruppe gruppe = new NormaleGruppe(dasRestaurant, ↵
gruppePersonen);
88             gruppe.IstGruppePlatziert = false;
89
90             listeDerGruppen.Add(gruppe);
91             dasRestaurant.ChefOber.GruppeKommt(gruppe);
92             _letzteGruppe = gruppe;
93             DebugNachrichtAnzeigen("Gruppe mit " + gruppePersonen.ToString() + ↵
" erstellt.");
94         }
95     }
96 }
97
98 [Serializable]
99 public class NormaleGruppe : EinfacheGruppe
100 {
101     public NormaleGruppe(Restaurant restaurant, int anzahlDerPersonen)
102     {
103         // Variablen initialisieren
104         _positionAmTisch = new GruppePositionAmTisch();
105         _positionAmTisch.GruppePositionStart = -1;
106         _bisherigeAufenthaltsdauer = 0;
107         _wartezzeitSitzplatz = 2;
108         _istGruppeSchonPlatziert = false;
109         _istGruppeSchonGegangen = false;
110         _personen = new List<IPerson>();
111
112         for (int i = 0; i < anzahlDerPersonen; i++)
113         {
114             // Erstelle Personen
115             _personen.Add(new Person("Normale Person"));
116         }
117
118         DasRestaurant = restaurant;
119     }
120
121     public NormaleGruppe()
122     {
123         // Variablen initialisieren
124         _positionAmTisch = new GruppePositionAmTisch();
125         _positionAmTisch.GruppePositionStart = -1;
126         _bisherigeAufenthaltsdauer = 0;
127         _wartezzeitSitzplatz = 2;
128         _istGruppeSchonPlatziert = false;
129         _istGruppeSchonGegangen = false;
130         _personen = new List<IPerson>();
131     }
132 }
```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\NormalePersonen.cs 3

```
133     public override bool IstSchuelerGruppe()
134     {
135         return false;
136     }
137
138     /// <summary>
139     /// Normale Personen habe eine andere "Farbe".
140     /// </summary>
141     public override System.Drawing.Color GuiGruppenFarbe
142     {
143         get
144         {
145             return Color.FromArgb(60, 60, 60);
146         }
147         set
148         {
149             base.GuiGruppenFarbe = value;
150         }
151     }
152 }
153 }
```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\KoordinatorGeld.cs 1
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using _28_2_bwinf_2.GUI;
6
7  namespace _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen
8  {
9      [Serializable]
10     public class KoordinatorGeld : EinfacherKoordinator
11     {
12         /// <summary>
13         /// Wird verwendet, um auf alle Elemente im Heap schnell (sequentiell)
14         /// zugreifen zu koennen, um den Kontostand der Personen zu veraendern.
15         /// </summary>
16         public List<PersonGeld> _listeDerPersonen;
17
18         public bool GUI_EinstellungsDialogEinfach = false;
19
20         public KoordinatorGeld()
21         {
22         }
23
24         public KoordinatorGeld(Restaurant restaurant)
25         {
26             KonstruktorEinArgument(restaurant);
27         }
28
29         public override string ToString()
30         {
31             return "KoordinatorGeld";
32         }
33
34         public override SteuerelementEinstellungen GuiEinstellungen()
35         {
36             SteuerelementEinstellungen fenster;
37
38             if (GUI_EinstellungsDialogEinfach)
39             {
40                 fenster = new EinfacherKoordinatorEinstellungen();
41             }
42             else
43             {
44                 fenster = new GeldKoordinatorEinstellungen();
45             }
46
47             fenster.DasObjekt = this;
48             return fenster;
49         }
50
51         public override int AnzahlDerPersonen
52         {
53             get
54             {
55                 return _anzahlDerPersonen;
56             }
57
58             set
59             {
60                 // Freie Personen erstellen
61                 FreiePersonen = new ObjectHeap<IPerson>();
62                 FreiePersonen.PersonenAusblenden = _personenAusblenden;
63                 _listeDerGruppen = new List<IGruppe>();
64                 _listeDerPersonen = new List<PersonGeld>();
65
66                 _anzahlDerPersonen = value;
67
68                 for (int i = 0; i < _anzahlDerPersonen; i++)
69                 {

```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\KoordinatorGeld.cs 2
70             PersonGeld aktuellePerson = new PersonGeld("Person " + i.ToString() ↵
71         ());
72         _listeDerPersonen.Add(aktuellePerson);
73     }
74 }
75 }
76
77 /// <summary>
78 /// Fuegt eine freie Person zur Liste (oder was auch immer) der freien Personen hinzu.
79 /// </summary>
80 protected override void FreiePersonHinzufuegen(IPerson person)
81 {
82     PersonGeld neuePerson = (PersonGeld) person;
83
84     // Wenn die Person kein Geld mehr hat, steht sie nicht mehr zur Verfuegung
85     if (neuePerson.Kontostand > 0)
86     {
87         FreiePersonen.FuegeElementEin(neuePerson);
88     }
89 }
90
91 /// <summary>
92 /// Entfernt eine freie Person aus der Liste der freien Personen.
93 /// </summary>
94 protected override IPerson FreiePersonHolen()
95 {
96     PersonGeld person = (PersonGeld) FreiePersonen.HoleErstesElement();
97     person.Kontostand--;
98
99     return person;
100 }
101 }
102
103 [Serializable]
104 public class PersonGeld : Person
105 {
106     public int Kontostand = 10;
107
108     public PersonGeld(string name)
109     {
110         Name = name;
111     }
112
113     /// <summary>
114     /// Vergleicht zwei PersonGeld-Klassen miteinander. Diese Klasse muss das IComparable-Interface wirklich implementieren, weil es vom Heap verwendet wird.
115     /// </summary>
116     public override int CompareTo(object obj)
117     {
118         PersonGeld personGeld = (PersonGeld) obj;
119         return Kontostand - personGeld.Kontostand;
120     }
121
122     public override string ToString()
123     {
124         return Name + " - " + Kontostand.ToString() + " Geldeinheiten";
125     }
126 }
127 }
128

```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\KellnerVerteilen.cs 1
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace _28_2_bwinf_2.RestaurantAlgorithmen.Erweiterungen
7 {
8     class KellnerVerteilen
9     {
10         /// <summary>
11         /// Erste Dimension: n, Zweite Dimension: t.
12         /// </summary>
13         private int[,] _matrixK;
14         private int[,] _matrixEntscheidung;
15
16         /// <summary>
17         /// Liste mit Gruppen (in der richtigen Reihenfolge), erstes Element hat den
18         Index 1.
19         /// </summary>
20         public List<int> _tischMitGruppen;
21         public int _tMax;
22         public int _nMax;
23
24         private List<int> _unterteilungen;
25         private int _kostenAktuellerFall;
26         private int _aktuelleVerschiebung;
27
28         public List<int> Start()
29         {
30             int besteKosten = int.MaxValue;
31             List<int> besteLoesung = new List<int>();
32             _aktuelleVerschiebung = 0;
33
34             for (int i = 0; i < _nMax; i++)
35             {
36                 GeneriereMatrixBasisfaelle();
37                 DerAlgorithmus();
38
39                 if (_kostenAktuellerFall < besteKosten)
40                 {
41                     besteKosten = _kostenAktuellerFall;
42                     besteLoesung = _unterteilungen;
43                 }
44
45                 // Verschiebe das Sitzplatzarray um einen Platz
46                 int erstes = _tischMitGruppen[1];
47                 _aktuelleVerschiebung++;
48
49                 for (int j = 1; j < _nMax; j++)
50                 {
51                     _tischMitGruppen[j] = _tischMitGruppen[j + 1];
52
53                     _tischMitGruppen[_nMax] = erstes;
54                 }
55
56                 return besteLoesung;
57             }
58
59             public void GeneriereMatrixBasisfaelle()
60             {
61                 // Matrix um 1 groesser machen, weil das erste Element den Index 0 hat
62                 _matrixK = new int[_nMax + 1, _tMax + 1];
63                 _matrixEntscheidung = new int[_nMax + 1, _tMax + 1];
64
65                 // n ist 1
66                 for (int i = 1; i <= _tMax; i++)
67                 {
68                     _matrixK[1, i] = _tischMitGruppen[1];
69                 }

```

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\KellnerVerteilen.cs 2
70
71         // t ist 1
72         for (int i = 1; i < _nMax + 1; i++)
73     {
74             int wert = 0;
75             // Aufsummieren
76             for (int j = 1; j < i + 1; j++)
77             {
78                 wert += _tischMitGruppen[j];
79             }
80
81             _matrixK[i, 1] = wert;
82         }
83     }
84
85     public void DerAlgorithmus()
86     {
87         _unterteilungen = new List<int>();
88
89         for (int n = 2; n < _nMax + 1; n++)
90     {
91         for (int t = 2; t < _tMax + 1; t++)
92     {
93             _matrixK[n, t] = int.MaxValue;
94
95             // Generiere alle Moeglichkeiten, den Trennstrich zu setzen
96             for (int i = 1; i < n + 1; i++)
97     {
98                 // Summe generieren
99                 int summe = 0;
100                for (int j = i + 1; j < n + 1; j++)
101                {
102                    summe += _tischMitGruppen[j];
103                }
104
105                int wert = Math.Max(_matrixK[i, t - 1], summe);
106
107                if (wert < _matrixK[n, t])
108                {
109                    // Es wurde eine bessere Loesung gefunden
110                    _matrixK[n, t] = wert;
111                    // Position der Teilung speichern
112                    _matrixEntscheidung[n, t] = i;
113                }
114            }
115        }
116    }
117
118    // Loesung auswerten
119    _kostenAktuellerFall = _matrixK[_nMax, _tMax];
120    int restgroesse = _nMax;
121    int unterteilungen = _tMax;
122
123    Console.WriteLine("Moegliche Loesung (Gruppengroesse):");
124    while (restgroesse != 0)
125    {
126        restgroesse = _matrixEntscheidung[restgroesse, unterteilungen];
127        unterteilungen--;
128
129        if (_aktuelleVerschiebung != 0)
130        {
131            unterteilungen.Add((restgroesse + _aktuelleVerschiebung - 1) % _nMax + 1);
132            Console.WriteLine((restgroesse + _aktuelleVerschiebung - 1) % _nMax + 1);
133        }
134        else
135        {
136            unterteilungen.Add(restgroesse);
137            Console.WriteLine(restgroesse);

```

F:\Users\meister\Documents\Visual Studio ...\\Erweiterungen\KellnerVerteilen.cs 3

```
138 }  
139 }  
140 }  
141 }  
142 }  
143 }
```

```
F:\Users\meister\Documents\Visual Studio ...\\Dateizugriff\LadenSpeichern.cs 1
1 using System.Collections.Generic;
2 using System.IO;
3 using System.Runtime.Serialization.Formatters.Binary;
4 using _28_2_bwinf_2.RestaurantAlgorithmen;
5
6 namespace _28_2_bwinf_2.Dateizugriff
7 {
8     static class LadenSpeichern
9     {
10         private static MemoryStream _restaurantImSpeicher;
11
12         /// <summary>
13         /// Speichert eine Instanz der Restaurant-Klasse und alle verknuepften Objekte in eine Datei.
14         /// </summary>
15         /// <param name="dateiname">Dateiname (Ausgabe)</param>
16         /// <param name="restaurant">Zeiger auf das Restaurant</param>
17         public static void RestaurantSpeichern(string dateiname, Restaurant restaurant)
18         {
19             restaurant.GuiEreignisseCallBackZeigerLoeschen();
20
21             FileStream schreiber = new FileStream(dateiname, FileMode.Create);
22             BinaryFormatter formatter = new BinaryFormatter();
23
24             formatter.Serialize(schreiber, restaurant);
25             schreiber.Flush();
26             schreiber.Close();
27
28             restaurant.GuiEreignisseSetzen();
29         }
30
31         /// <summary>
32         /// Speichert eine Kopie des Restaurants im RAM.
33         /// </summary>
34         public static void RestaurantRAMSpeichern(Restaurant restaurant)
35         {
36             restaurant.GuiEreignisseCallBackZeigerLoeschen();
37
38             _restaurantImSpeicher = new MemoryStream();
39             BinaryFormatter formatter = new BinaryFormatter();
40
41             formatter.Serialize(_restaurantImSpeicher, restaurant);
42             _restaurantImSpeicher.Flush();
43
44             restaurant.GuiEreignisseSetzen();
45         }
46
47         /// <summary>
48         /// Laedt ein Restaurant aus dem RAM.
49         /// </summary>
50         public static Restaurant RestaurantRAMLaden()
51         {
52             BinaryFormatter formatter = new BinaryFormatter();
53             // An den Anfang des Streams springen
54             _restaurantImSpeicher.Seek(0, SeekOrigin.Begin);
55             Restaurant rueckgabe = (Restaurant) formatter.Deserialize
56             (_restaurantImSpeicher);
57             _restaurantImSpeicher.Close();
58
59             rueckgabe.NachDeserialisierung();
60             return rueckgabe;
61         }
62
63         /// <summary>
64         /// Laedt eine Instanz der Restaurant-Klasse aus einer Datei.
65         /// </summary>
66         /// <param name="dateiname">Dateiname (Eingabe)</param>
67         /// <returns>Zeiger auf das Restaurant</returns>
68         public static Restaurant RestaurantLaden(string dateiname)
69         {
```

```
F:\Users\meister\Documents\Visual Studio ...\\Dateizugriff\LadenSpeichern.cs 2
69     FileStream leser = new FileStream(dateiname, FileMode.Open);
70     BinaryFormatter formatter = new BinaryFormatter();
71
72     Restaurant rueckgabe = (Restaurant) formatter.Deserialize(leser);
73     leser.Close();
74
75     rueckgabe.NachDeserialisierung();
76     return rueckgabe;
77
78 }
79 }
80 }
81 }
```