Programmierpraktikum (Java)
Dr. Vera Weil

Institut für Informatik, Universität zu Köln
Sommersemester 2018

**MEILENSTEIN 1**

---

# Inhaltsverzeichnis

# 1 Allgemeine und organisatorische Hinweise

💻 = http://informatik.uni-koeln.de/weil/

## 1.1 Rund um die Theorieaufgaben

- **Deadline:** Abgabe der Theorieaufgaben bis spätestens **30.05.2018 um 12:00 Uhr**.
- **Schriftliche Abgaben** unbedingt mit Name, Matrikelnummer und Gruppennummer („team40“, zum Beispiel) versehen, sonst werden die Abgaben **nicht** berücksichtigt. Abgabe ist nur möglich über Briefkasten Weyertal 121, 5. Etage. Scans, Fotos oder Ähnliches sind nicht erlaubt und werden gleich gelöscht.
- Beachten Sie bitte, dass ein Teil der Theorieaufgaben daraus bestehen wird, **Javadoc**-taugliche Kommentare für all Ihre Methoden zu schreiben. Wenn Sie diese Kommentare schon während der Erstellung Ihres Programms anständig pflegen, spart Ihnen das viel Arbeit. Zudem ist die damit erstellte Dokumentation vielleicht auch für Sie schon während der Bearbeitung der Aufgaben hilfreich. Probieren Sie es aus!

## 1.2 Rund um die Programmieraufgaben

- **Deadline:** Abgabe der Programmieraufgaben bis spätestens **30.05.2018 um 12:00 Uhr**.
- **Sonderzeichen**: Beachten Sie bitte, dass es mit Umlauten und ß zu Problemen kommen kann. Ersetzen Sie also ä,ü,ö und ß mit ae, ue, oe und ss, um sicher zu gehen, dass Ihr Programm auf unserem Server genauso läuft wie bei Ihnen. Das gilt natürlich auch für die Kommentare.
- Achten Sie **genauestens** auf die Vorgaben, vor allem auf die Schreibweise der Methoden. Wir machen eine automatische Korrektur. Das bedeutet, dass der PC mit Cornercases testet. Sobald Ihr Lösungs**output** von der unsrigen um nur „ein Epsilon“ abweicht, gibt es dafür 0 Punkte.
- Auf unserer Internetseite stellen wir Ihnen die Klasse `Grid` und die Klasse `Cell` zur Verfügung. Diese bieten Ihnen die **Datenstrukturen**, auf denen die Interfaces basieren.
- **Abgabe der individuellen Programmieraufgaben**: Die genauen Modalitäten werden auf der Website bekannt gegeben.
- Projektformat Meilenstein 1: Ihr Projekt muss folgende Namensgebung und Struktur aufweisen:

```
|-- javadoc
`-- src
   |-- App.java
   |-- data
   |    |-- Cell.java
   |    `-- Grid.java
   `-- utils
```

  - Die Datei `App.java` dürfen Sie so verwenden, dass dort eine ausführbare Datei entsteht, mit der Sie testen können, etc. Wir werden `App.java` mit unserer eigenen Datei (mit der wir Ihr Programm testen) überschreiben. Was da also drin steht, wird nicht bewertet.

  - `Cell.java` und `Grid.java` stellen wir Ihnen online (💻) zur Verfügung.
  - Der Ordner `utils` enthält sowohl Ihre Interfaces als auch die von Ihnen implementierten Klassen.
  - Im Ordner `javadoc` ist die Javadoc-HTML-Datei zu speichern.
  - Wenn Sie Ihr Projekt abgeben möchten, dann packen Sie es bitte als zip - Datei mit Namen `sudoku-src.zip` und laden diese exportierte Datei auf der entsprechenden Internetseite hoch. Wie die genaue `url` dieser Seite lautet, ab wann sie online ist und weitere Modalitäten werden auf der Internetseite noch bekannt gegeben.

- Beachten Sie bitte, dass Sie die **Gruppenaufgabe via Git** abgeben müssen. Daher empfiehlt es sich bereits bei den Individualaufgaben, mit **Git** zu arbeiten, auch wenn Sie diese über unsere Abgabewebseite hochladen. Genauere Informationen zum Abgeben Ihrer Lösungen veröffentlichen wir bald auf 💻.

- Wir arbeiten noch an an den Git-Repositories, die Sie für Ihre Individualaufgaben benutzen dürfen und für Ihre Gruppenaufgaben benutzen müssen. Die Git-Repositories sollten aber Ende der Woche auf unserem Server bereit sein. Näheres auf der Website (💻).

- Die von Ihnen hochgeladenen Lösungen dürfen auf **gar keinen Fall Ausgaben machen** (auf der Konsole, zum Beispiel). Beachten Sie das bei der Erstellung Ihrer Programme. Vermutlich werden Sie testweise Ausgaben machen lassen, denken Sie aber daran, dass in der abgegebenen Version zu unterbinden. Sonst wird es bei Ihrem Programm Probleme bei der automatisierten Korrektur geben!

- Welche **Bibliotheken, Pakete, Klassen** darf ich benutzen? Sie dürfen alles „innerhalb“ der JDK verwenden. Das bedeutet, alle in der `Java`$^{TM}$ `Platform Standard Ed. 8` aufgeführten Elemente (Klassen, Pakete...) dürfen Sie verwenden.
  Anders gesagt dürfen Sie das, was hier aufgeführt wird, verwenden:
  `https://docs.oracle.com/javase/8/docs/api/index.html?overview-tree.html`
  Plus natürlich noch das, was wir Ihnen explizit zur Verfügung stellen.
  Die Benutzung externer Frameworks und Bibliotheken ist Ihnen allerdings untersagt.

- Wir gehen davon aus, dass Sie **Java Standard Edition 8** verwenden. Auf unserem Server verwenden wir das `Oracle Java SE Development Kit 8u171`, kurz `Oracle JDK 8`.

## 1.3 Sonstiges

- Wenn Sie die **offenen Fragestunden** in Anspruch nehmen möchten, so bringen Sie, wenn möglich, Ihren eigenen Laptop mit.

- Lassen Sie sich von der **Fülle der Aufgaben** nicht abschrecken. Viele Aufgaben sind einander ähnlich, so dass, wenn Sie eine gelöst haben, oft „nicht mehr viel“ fehlt, um ähnliche Aufgaben zu lösen.

- Lassen Sie sich von der **Menge an Definitionen und Hinweisen** nicht abschrecken. Die meisten davon sind ohnehin „intuitiv“, mussten von mir aber formalisiert werden, damit die Aufgabenstellungen eindeutig sind.

- Beachten Sie bitte, dass besonders die Interfaces in den Individualaufgaben sicherlich **keine sinnvolle Aufteilung im Sinne des Software Engineering** sind. Sie ist mehr dem Fakt geschuldet, dass wir jeweils vergleichbare Aufgaben stellen wollten.

- Wenn Ihnen auf diesem Blatt **Inkonsistenzen oder Unklarheiten** auffallen, so fragen Sie nach. Nach Abgabe Ihrer Programme darauf aufmerksam zu machen, bringt niemandem etwas und wird dann auch in der Korrektur nicht mehr berücksichtigt werden.
- Die Sudokus wurden mit Hilfe der LaTeX- Beispiele von Roberto[1] Bonvallet erstellt.
- Besonderen Dank gilt Philipp Klinke, der sowohl die Accounts und Abgabemodalitäten verwaltet als auch bei der Erstellung der Aufgaben eine große Hilfe war und mir als (ursprünglich) Mathematikerin die Sicht der Wirtschaftsinformatikerinnen und Wirtschaftsinformatiker näher brachte. Meilenstein 2 kann kommen!
- Für einige gute Anregungen und Ideen danke ich meinem Programmierpraktikum-Tutoren-Team: Felix, Karl, Marvin, Tim und Philippe. Macht Euch bereit für Meilenstein 2!

# 2 Begriffe und Definitionen

## 2.1 Gitter, Sudoku, Zelle und Einheit

- Mit **Gitter** bezeichnen wir ein Quadrat, dessen Fläche wiederum in gleichgroße Quadrate unterteilt ist. Ein kleinstes, nicht weiter unterteiltes Quadrat bezeichnen wir als **Zelle**.

  ▷ Beispiel: Abbildung 1.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | | 5 | | 1 | | 9 | |
| 8 | | | 2 | | 3 | | | 6 |
| | 3 | | | 6 | | | 7 | |
| | | 1 | | | | 6 | | |
| 5 | 4 | | | | | | 1 | 9 |
| | | 2 | | | | 7 | | |
| | 9 | | | 3 | | | 8 | |
| 2 | | | 8 | | 4 | | | 7 |
| | 1 | | 9 | | 7 | | 6 | |

Abbildung 1: Gitter, das in 81 Zellen unterteilt ist.

- Sei $n \in \mathbb{N}$. Mit **n x n - Gitter** bezeichnen wir ein Gitter, welches genau $n^2$ viele Zellen enthält. Automatisch verteilen sich diese Zellen auf n untereinander stehende **Zeilen**, welche aus jeweils n nebeneinander stehenden Zellen bestehen. Analog lässt sich das Gitter betrachten als n nebeneinander stehende **Spalten**, welche aus jeweils n untereinander stehenden Zellen bestehen.

  ▷ Beispiel: In Abbildung 1 ist ein $9 \times 9$-Gitter abgebildet. Somit gibt es dort 9 untereinander stehende Zeilen und 9 nebeneinander stehende Spalten.

- Die Zellen eines $n \times n$ - Gitters werden mit $z_{i,j}$, $i, j \in \{1, \ldots, n\}$ bezeichnet. Dabei entspricht $z_{i,j}$ der $j$-ten-Zelle in der $i$-ten Zeile. Wir bezeichnen in diesem Falle $i$ als **Zeilenindex** und $j$ als **Spaltenindex**.

  ▷ Beispiele: In Abbildung 2 haben wir Ihnen zur Verdeutlichung mal ein paar Bezeichnungen eingetragen.

[1] In Version vorher stand hier: Robert

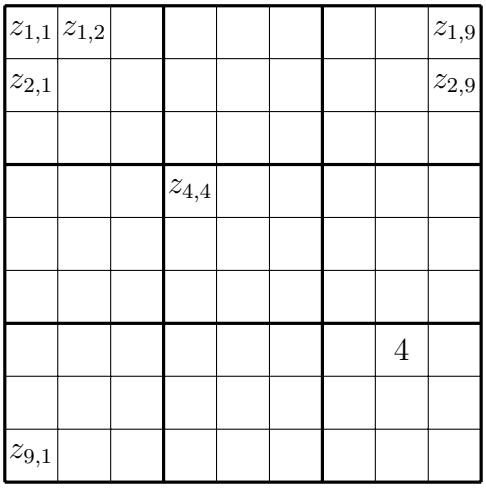


Abbildung 2: Gitter mit ein paar Zellenbezeichnungen. Die Zelle, in der die 4 steht, hat als Spaltenindex 8 und als Zeilenindex 7.

- Mit **Sudoku** bezeichnen wir auf diesem Aufgabenblatt ein $9 \times 9$ - Gitter.
- Ein Sudoku lässt sich eindeutig in neun $3 \times 3$ - Gitter partitionieren. Ein solches $3 \times 3$-Gitter nennen wir **Block**.
  - ▷ Beispiel: Ein Block wird zum Beispiel von den Zellen $z_{1,1}$, $z_{1,2}$, $z_{1,3}$, $z_{2,1}$, $z_{2,2}$, $z_{2,3}$, $z_{3,1}$, $z_{3,2}$ und $z_{3,3}$ gebildet.
- Als **Einheit** bezeichnen wir entweder eine Zeile, eine Spalte oder einen Block eines Sudokus.
  - ▷ Hinweis: Dies dient lediglich dazu, abkürzende Schreibweisen zu verwenden.
- Wir sagen, dass eine Zelle A **kleiner** als eine Zelle B ist, falls entweder der Zeilenindex von A kleiner ist als der Zeilenindex von B, oder falls der Zeilenindex von A gleich dem Zeilenindex von B ist und zugleich der Spaltenindex von A kleiner als der Spaltenindex von B ist.
  - ▷ Beispiele: Die Zelle $z_{1,1}$ ist kleiner als die Zelle $z_{1,3}$. Die Zelle $z_{1,8}$ ist kleiner als die Zelle $z_{2,1}$.
  - ▷ Hinweis: Anschaulich gesagt ist A kleiner als B, falls A entweder weiter oben im Gitter ist als B, oder falls A genauso weit oben im Gitter zu finden ist wie B, aber weiter links im Gitter. Es ist dabei im Übrigen unerheblich, welchen *Wert* die Zelle in sich trägt.
- Wir nennen eine Zelle **minimal**, wenn es keine kleinere Zelle gibt.
  - ▷ Beispiele: Siehe Abbildung 3.
- Die **Ankerzelle** einer Einheit ist die minimale Zelle einer Einheit.
  - ▷ Beispiele: Siehe Abbildung 3.
- Sei $P$ eine Menge von Zellpaaren. Aus jedem Zellpaar suchen wir uns die kleinere Zelle heraus. Unter diesen kleineren Zellen bestimmen wir die minimale Zelle, sagen wir $Z$. Unter allen Partnern von $Z$, die aus $P$ zu entnehmen sind, wählen wir wieder die minimale Zelle, sagen wir $U$. Dann bilden $Z$ und $U$ das **minimale Zellpaar** aus $P$.

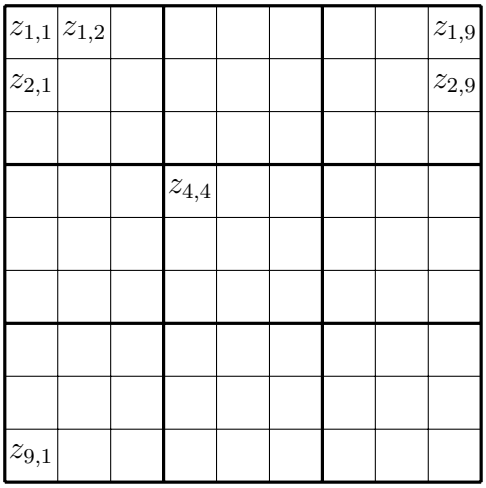


Abbildung 3: Die Zelle $z_{4,4}$ ist in dem mittleren Block minimal und somit die Ankerzelle des Blocks. Die Zelle $z_{9,1}$ ist in der untersten Zeile minimal und somit die Ankerzelle der Zeile. Die Zelle $z_{1,2}$ ist in der zweit-linkesten Spalte minimal und somit die Ankerzelle der Spalte.

- ▷ Hinweis: Anschaulich gesagt möchten wir unter den Zellpaaren mit einer gewissen Eigenschaft ein minimales festlegen können. Wir suchen uns unter all diesen Zellpaaren die Paare raus, die die minimale Zelle beinhalten. Unter diesen Zellpaaren nehmen wir dann den Partner der minimalen Zelle, der selbst wieder minimal ist unter allen Partnern der minimalen Zelle.
- ▷ Beispiel: Sei $P$ die Menge der Paare $\{z_{1,1}, z_{2,2}\}$, $\{z_{3,1}, z_{1,3}\}$ und $\{z_{1,1}, z_{1,3}\}$. Die Menge der kleineren Zellen ist folglich $\{z_{1,1}, z_{1,3}\}$. Die minimale Zelle dieser Menge ist $z_{1,1}$. Die Menge aller Partner von $z_{1,1}$ ist $\{z_{2,2}, z_{1,3}\}$. In dieser Menge ist die minimale Zelle $z_{1,3}$. Das minimale Zellpaar ist folglich $\{z_{1,1}, z_{1,3}\}$.
- ▷ Weiteres Beispiel: Siehe Abbildung 4.

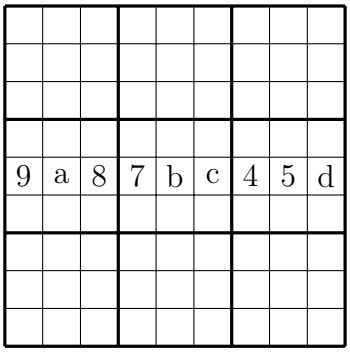


Abbildung 4: Minimale Zellpaare: Alle Zellpaare mit der Eigenschaft, dass Buchstaben in den Zellen geschrieben worden sind, sind die Paare $\{z_{5,2}, z_{5,5}\}$, $\{z_{5,2}, z_{5,6}\}$, $\{z_{5,2}, z_{5,9}\}$, $\{z_{5,5}, z_{5,6}\}$, $\{z_{5,5}, z_{5,9}\}$ und $\{z_{5,6}, z_{5,9}\}$. Unter den Zellen in diesen Zellpaaren suchen wir die minimale Zelle. Das ist $z_{5,2}$. Unter allen Zellen, die mit $z_{5,2}$ in einem Zellpaar enthalten sind, suchen wir die minimale Zelle. Das ist dann $z_{5,5}$. Das minimale Zellpaar ist folglich $\{z_{5,2}, z_{5,5}\}$.

## 2.2 Belegungen

- Wir nennen eine Zelle **positiv belegt**, wenn sie eine Zahl zwischen 1 und 9 enthält. Wir nennen eine Zelle **leer**, wenn sie mit der Zahl `-1` belegt ist.

▷ Hinweis: Auf den meisten Abbildungen wird eine mit -1 belegte Zelle als „ganz weiße“ Zelle dargestellt. In unserem Programm werden wir leere Zellen mit der Zahl `-1` füllen, um sie als leere Zelle zu markieren.

▷ Beispiel: Die linkeste Zelle des Gitters in Abbildung 1 ist eine leere Zelle.

- Eine Einheit, in der nur Zahlen aus $\{-1, 1, \ldots, 9\}$ enthalten sind und in der jede natürliche Zahl zwischen 1 und 9 nur höchstens einmal vorkommt, nennen wir **zulässig belegt**.

  ▷ Hinweis: Anschaulich bedeutet das, dass zum Beispiel eine Zeile, die wir als zulässig belegt bezeichnen, höchstens einmal die Zahl 1, höchstens einmal die Zahl 2, und so weiter enthalten darf, und dementsprechend höchstens neun leere Zellen.

  ▷ Beispiel: Jede Einheit aus dem Sudoku in Abbildung 1 ist zulässig belegt.

- Ein Sudoku ist **zulässig belegt**, wenn jede Einheit des Sudokus zulässig belegt ist.

  ▷ Beispiel für ein zulässig belegtes Sudoku: Siehe Abbildung 1.

- Ein Sudoku ist **gelöst**, falls es zulässig belegt ist und keine leeren Zellen beinhaltet.

## 2.3 Abbildungen

- Die **n.-te Stelle** in einem Array ist der n.-te Speicherplatz, der in einem Array belegt werden kann.

  ▷ Beispiel: Array `a` sei durch folgenden Befehl belegt worden: `int[] a = {5,32,7};` Dann steht die 5 an der ersten Stelle des Arrays (obwohl der Zellen**index** 0 ist).

- Eine **Wertpermutation auf einem Sudoku** ist eine Abbildung der Zahlen $\{-1, 1, \ldots, 9\}$ in die Zahlen $\{-1, 1, \ldots, 9\}$ mit folgenden Eigenschaften:

  1. Die `-1` wird immer auf die `-1` abgebildet.

     ▷ Hinweis: Die -1 steht für die nicht gefüllten Zellen. Anders gesagt: Leere Zellen dürfen nur wieder auf leere Zellen abgebildet werden.

  2. Die Zahlen $\{1, \ldots, 9\}$ werden in die Zahlen $\{1, \ldots, 9\}$ abgebildet. Dabei müssen alle Zahlen aus $\{1, \ldots, 9\}$, die in dem Sudoku auftreten, auf eine Zahl aus $\{1, \ldots, 9\}$ abgebildet werden.

     ▷ Hinweis: Alle vorkommenden Zahlen dürfen also nicht auf eine -1 abgebildet werden. Anders gesagt: Eine nicht leere Zelle darf nur auf eine nicht leere Zelle abgebildet werden.

  3. Wird eine Zahl $i$ aus $\{1, \ldots, 9\}$ auf eine Zahl $j$ in $\{1, \ldots, 9\}$ abgebildet, so darf keine weitere Zahl aus $\{1, \ldots, 9\}$ auf $j$ abgebildet werden.

     ▷ Beispiel: Wird die 9 auf die 1 abgebildet, darf die 4 nicht auch auf die 1 abgebildet werden.

     ▷ Hinweis und Beispiele: Wie kann man sich eine Wertpermutation vorstellen? Hier geht es darum, dass wir Zahlen „austauschen“, leere Zellen aber leere Zellen bleiben. Siehe dazu die Abbildungen 5 und 6.

- Eine **Wertpermutation auf einer Einheit** wird analog definiert. Ersetzen Sie in der Definition die Worte „in/auf einem Sudoku“ mit „auf/in einer Einheit“.

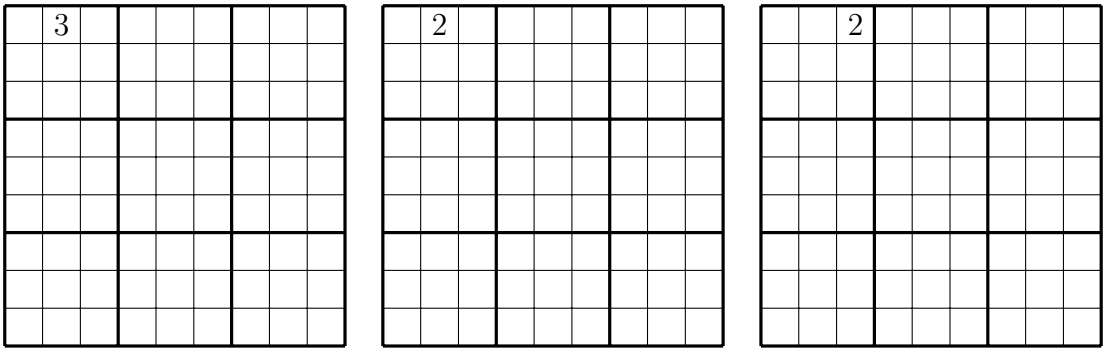


Abbildung 5: Das mittlere Gitter ist durch eine Wertpermutation aus dem linken Gitter entstanden. Alle Zahlen zwischen 1 und 9 sind auf die -1 abgebildet worden, mit Ausnahme der 3. Die 3 ist auf die 2 abgebildet worden. Das rechte Gitter ist nicht durch eine Wertpermutation auf dem linken Gitter entstanden: Die -1 aus $z_{1,3}$ ist auf eine 2 abgebildet worden und widerspricht so den vorgegebenen Eigenschaften. Genauso kann man argumentieren, dass die 3 aus $z_{1,2}$ auf die -1 abgebildet worden ist. Auch dies widerspricht den vorgegebenen Eigenschaften.

- Der **Bildvektor** einer Wertpermutation ist eine neunstellige, geordnete Zahlenmenge

  $$(a_1, \ldots, a_9)$$

  mit der Eigenschaft, dass $a_i$, $i \in \{1, \ldots, 9\}$, angibt, auf welchen Wert $i$ abgebildet wird.

  ▷ Beispiel: Die Wertpermutation, die in Abbildung 5 beschrieben wird, hat den folgenden Bildvektor:

  $$(-1, -1, 2, -1, -1, -1, -1, -1, -1)$$

  Daran kann man ablesen, dass die 3 auf die 2 abgebildet wurde, und alle anderen Zahlen zwischen 1 und 9 in dem ursprünglichen linken Gitter gar nicht vorkommen.

  ▷ Beispiel: Die Wertpermutation, die in Abbildung 6 beschrieben wird, hat den folgenden Bildvektor:

  $$(2, 3, 4, 5, 6, 7, 8, 9, 1)$$

  Daran kann man beispielsweise ablesen, dass die 4 auf die 5 abgebildet wird, weil die 5 an vierter Stelle des Bildvektors steht.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | | 5 | | 1 | | 9 | |
| 8 | | | 2 | | 3 | | | 6 |
| | 3 | | | 6 | | | 7 | |
| | | 1 | | | | 6 | | |
| 5 | 4 | | | | | | 1 | 9 |
| | | 2 | | | | 7 | | |
| | 9 | | | 3 | | | 8 | |
| 2 | | | 8 | | 4 | | | 7 |
| | 1 | | 9 | | 7 | | 6 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 3 | | 6 | | 2 | | 1 | |
| 9 | | | 3 | | 4 | | | 7 |
| | 4 | | | 7 | | | 8 | |
| | | 2 | | | | 7 | | |
| 6 | 5 | | | | | | 2 | 1 |
| | | 3 | | | | 8 | | |
| | 1 | | | 4 | | | 9 | |
| 3 | | | 9 | | 5 | | | 8 |
| | 2 | | 1 | | 8 | | 7 | |

Abbildung 6: Das rechte Gitter ist aus dem linken Gitter entstanden, nachdem eine Wertpermutation auf dem linken Gitter ausgeführt wurde. Hier wurde die Zahl $i$ auf die Zahl $i + 1$, für $1 \leq i \leq 8$, abgebildet; sowie die Zahl 9 auf die Zahl 1.

- Wir sagen, dass eine Einheit B **durch die Anwendung einer Wertpermutation** auf eine Einheit A entstanden ist, falls es eine Wertpermutation gibt, welche die Zahl $i$ aus $A$

auf die Zahl $a_i$ aus $B$ abbildet. Dementsprechend ist der Bildvektor der Wertpermutation $(a_1, \ldots, a_9)$.

- Eine **blockinterne Spaltenpermutation** ist eine Spaltenpermutation innerhalb eines Blocks A, wobei mit Spalte tatsächlich die ganze, 9-zellige Spalte gemeint ist. Die sechs Spalten außerhalb des Blocks A bleiben dabei unberührt. Zu jeder blockinternen Spaltenpermutation gehört eine geordnete Dreiermenge an Ganzzahlen $(a, b, c)$. Dieser **Bildvektor** $(a, b, c)$ ist so zu verstehen, dass in dem zu permutierenden Block die erste Spalte auf die a-te Spalte des Blocks, die zweite Spalte auf die b-te Spalte des Blocks und die dritte Spalte auf die c-te Spalte des Blocks abgebildet wird.

  ▷ Hinweis: Eine blockinterne Spaltenpermutation des Blocks mit Ankerzelle $z_{1,1}$ ist gleichzeitig eine blockinterne Spaltenpermutation des Blocks mit Ankerzelle $z_{4,1}$ und ist gleichzeitig eine blockinterne Spaltenpermutation des Blocks mit Ankerzelle $z_{7,1}$.

  ▷ Beispiel: Siehe Abbildung 7.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | | 1 | | | | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| 1 | | 9 | | 8 | | | | |
| | | 8 | | 9 | 1 | | | |
| 5 | 7 | 6 | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | 1 | | | | | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| | 9 | 1 | | 8 | | | | |
| | 8 | | | 9 | 1 | | | |
| 7 | 6 | 5 | | | | | | |

Abbildung 7: Dieses Sudoku ist das Ergebnis einer blockinternen Spaltenpermutation im Block mit Ankerzelle $z_{1,1}$ auf dem Sudoku aus Abbildung 15 (das rechte Sudoku). Der Bildvektor ist in diesem Fall $(2, 3, 1)$, weil die vormals erste Spalte nun die zweite Spalte in dem Block ist, die vormals zweite Spalte nun die dritte Spalte in dem Block ist, und die vormals dritte Spalte nun die erste Spalte in dem Block ist.

- Eine **blockinterne Zeilenpermutation** ist eine Zeilenpermutation innerhalb eines Blocks A, wobei mit Zeile die ganze, 9-zellige Zeile gemeint ist. Die sechs Zeilen außerhalb des Blocks A bleiben dabei unberührt. Zu jeder blockinternen Zeilenpermutation gehört eine geordnete Dreiermenge an Ganzzahlen $(a, b, c)$. Dieser **Bildvektor** $(a, b, c)$ ist so zu verstehen, dass in dem zu permutierenden Block die erste Zeile auf die a-te Zeile des Blocks, die zweite Zeile auf die b-te Zeile des Blocks und die dritte Zeile auf die c-te Zeile des Blocks abgebildet wird.

  ▷ Hinweis: Es gilt das Analogon zur blockinternen Spaltenpermutation.

- Eine **horizontale Blockpermutation** ist eine Spaltenpermutation, bei der die Reihenfolge der Spalten mit Ankerzelle $z_{1,1}, z_{1,2}, z_{1,3}$, die Reihenfolge der Spalten mit Ankerzelle $z_{1,4}, z_{1,5}, z_{1,6}$ und die Reihenfolge der Spalten mit Ankerzelle $z_{1,7}, z_{1,8}, z_{1,9}$ unberührt bleibt.

  ▷ Beispiel: Entstand Gitter2 dardurch, dass aus Gitter1 Spalte 1 auf Spalte 4, Spalte 2 auf Spalte 5, Spalte 3 auf Spalte 6, Spalte 4 auf Spalte 1, Spalte 5 auf Spalte 2, Spalte 6 auf Spalte 3 verschoben und blieb Spalte 7 auf Spalte 7, Spalte 8 auf Spalte 8, Spalte 9 auf Spalte 9, so handelt es sich hierbei um eine horizontale Blockpermutation.

▷ Beispiel: Entstand Gitter2 dardurch, dass aus Gitter1 Spalte 1 auf Spalte 4, Spalte 2 auf Spalte 6, Spalte 3 auf Spalte 5, Spalte 4 auf Spalte 1, Spalte 5 auf Spalte 2, Spalte 6 auf Spalte 3 verschoben und blieb Spalte 7 auf Spalte 7, Spalte 8 auf Spalte 8, Spalte 9 auf Spalte 9, so handelt es sich hierbei nicht um eine horizontale Blockpermutation, da es zu einer Vertauschung innerhalb der ersten drei Spalten gekommen ist.

▷ Beispiel: Siehe Abbildung 8.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | | 1 | | | 1 | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| 1 | | 9 | | 8 | | | | |
| | | 8 | | 9 | 1 | | | |
| 5 | 7 | 6 | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | |
| | | | | | | 1 | | 9 |
| | | 1 | | | 1 | | | |
| | 1 | | | | | | | |
| | | | | | | | 1 | 8 |
| | 8 | | 1 | | 9 | | | |
| | 9 | 1 | | | 8 | | | |
| | | | 5 | 7 | 6 | | | |

Abbildung 8: Das rechte Sudoku ist das Ergebnis einer horizontale Blockpermutation auf dem linken Sudoku. Die drei Spalten links sind in die Mitte verschoben worden, die drei Spalten in der Mitte sind nach links verschoben worden. Der Blockbildvektor ist in diesem Fall $(z_{1,4}, z_{1,1}, z_{1,7}, z_{4,4}, z_{7,4})$.

- Eine **vertikale Blockpermutation** definiert sich analog zu einer horizontalen Blockpermutation.
- Eine **Blockpermutation** ist eine horizontale Blockpermutation oder vertikale Blockpermutation.
- Zu jeder Blockpermutation existiert eine fünfstellige, geordnete Menge an Indexpaaren. Diese Menge nennen wir **Blockbildvektor**. Ist Grid B durch Anwendung einer Blockpermutation auf Grid A entstanden, so ist $z_{1,1}$ von A auf Zelle $z_{x,y}$ von B abgebildet worden. Analog ist $z_{1,4}$ auf $z_{a,b}$, $z_{1,7}$ auf $z_{c,d}$, $z_{4,1}$ auf $z_{e,f}$, und $z_{7,1}$ auf $z_{g,h}$ abgebildet worden. Der **Blockbildvektor** wird als ein fünfstelliges Array von Zellen in der Reihenfolge

  $$(z_{x,y}, z_{a,b}, z_{c,d}, z_{e,f}, z_{g,h})$$

  dargestellt. Das bedeutet also zum Beispiel, dass an der dritten Zelle des Arrays steht, auf welche Zelle in B die Zelle $z_{1,7}$ von A abgebildet wurde.

  ▷ Beispiel: Siehe Abbildung 8.

- Die **Transposition eines Gitters** ist das Abbilden des Wertes aus Zelle $z_{i,j}$ in die Zelle $z_{j,i}$, für alle $i, j \in \{1, \ldots, 9\}$. Wird eine Transposition auf ein Gitter angewandt, so wird das Gitter **transponiert**.

  ▷ Hinweis: Grob gesagt ist eine Transposition eine Spiegelung der Werte an der Diagonalen, die von oben links nach unten rechts im Gitter verläuft.

  ▷ Beispiel: Siehe Abbildung 9.

- Bei der **Spiegelung eines Gitters** an der Spalte mit Spaltenindex 5 wird

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | | 1 | | | 1 | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| 1 | | 9 | | 8 | | | | |
| | | 8 | | 9 | 1 | | | |
| 5 | 7 | 6 | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | 1 | | 5 |
| | | | | | | | | 7 |
| | | | 1 | | | 9 | 8 | 6 |
| | 1 | | | | | | | |
| | | | | 1 | | 8 | 9 | |
| | | | 1 | | | | 1 | |
| | | 1 | | | | | | |
| | | | | | 1 | | | |
| | | 9 | | | 8 | | | |

Abbildung 9: Das rechte Sudoku ist das Ergebnis einer Transposition des linken Sudokus.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | | 1 | | | 1 | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| 1 | | 9 | | 8 | | | | |
| | | 8 | | 9 | 1 | | | |
| 5 | 7 | 6 | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | | | |
| 9 | | 1 | | | | | | |
| | | | 1 | | | 1 | | |
| | | | | 1 | | | | |
| 8 | 1 | | | | | | | |
| | | | | 8 | | 9 | | 1 |
| | | | 1 | 9 | | 8 | | |
| | | | | | | 6 | 7 | 5 |

Abbildung 10: Das rechte Sudoku ist das Ergebnis einer Spiegelung des linken Sudokus an der Spalte mit Spaltenindex 5.

1. die Spalte mit Spaltenindex 5 unberührt belassen,
2. jede Spalte mit Spaltenindex größer als 5 wird auf eine Spalte mit Spaltenindex kleiner als 5 abgebildet. Dabei muss beachtet werden, dass die Spalte mit größtem Spaltenindex auf die Spalte mit kleinstem Spaltenindex abgebildet wird. Die Spalte mit zweitgrößtem Spaltenindex wird auf die Spalte mit zweitkleinstem Spaltenindex abgebildet, und so weiter. Die Zeilenindices bleiben unberührt.

Bei der Spiegelung eines Gitters an der Zeile mit Zeilenindex 5 ersetzen Sie bei den obigen Spiegelstrichen das Wort Spalte mit Zeile und umgekehrt.

▷ Beispiel: Siehe Abbildung 10.

- Seien $D_1$ die Zellen des Dreiecks, das durch die Diagonale beginnend bei $z_{1,1}$ und endend bei $z_{9,9}$, der Zeile beginnend bei $z_{1,1}$ und der Spalte beginnend bei $z_{1,9}$ begrenzt wird, inklusive der Zellen auf dem Rand des Dreiecks. Seien $D_2$ die Zellen des Dreiecks, das durch die Diagonale beginnend bei $z_{1,1}$ und endend bei $z_{9,9}$, der Spalte beginnend bei $z_{1,1}$ und der Zeile beginnend bei $z_{9,1}$ begrenzt wird, inklusive der Zellen auf dem Rand des Dreiecks. Die **Rechtsdrehung eines Gitters** verschiebt die Werte des gesamten Dreiecks $D_1$ und $D_2$ so, dass der Wert in Zelle $z_{1,1}$ in die Zelle $z_{1,9}$ und Zelle $z_{9,9}$ in die Zelle $z_{9,1}$ abgebildet werden. Dabei wird die relative Position des Wertes in einem Dreieck nicht verändert. *War also eine Zahl vorher zu den den Zahlen $a_1, a_2, a_3$ und $a_4$ benachbart, so ist dies nach der Rechtsdrehung immer noch so.*

  ▷ Hinweis: Stellen Sie sich einfach vor, Sie schneiden das Gitter aus, drehen es eine Vierteldrehung nach rechts (also die Ecke links oben auf die Ecke rechts oben). Die jetzt liegenden Zahlen $\overline{1}$ bis $\overline{9}$ richten Sie einfach wieder auf, so dass 1 bis 9 wieder

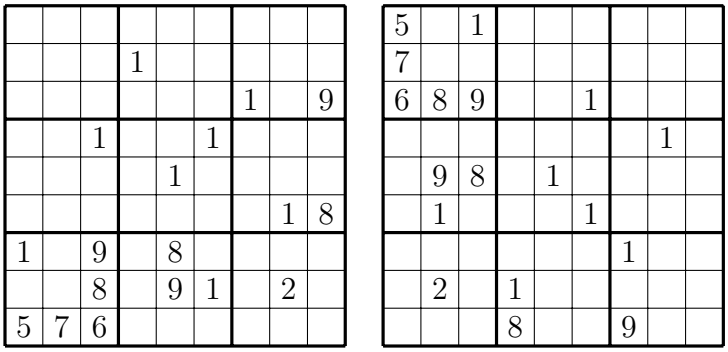

Abbildung 11: Das rechte Sudoku ist das Ergebnis einer Rechtsdrehung des linken Sudokus.

stehen. Das daraus resultierende Gitter ist das Gitter, das nach Rechtsdrehung aus dem ursprünglichen Gitter entstanden ist.

▷ Beispiel: Siehe Abbildung 11.

## 2.4 Begriffe aus Lösungsstrategien

- Eine **FullHouse-Einheit** ist eine zulässig belegte Einheit, die 8 natürliche Zahlen enthält.

  ▷ Hinweis: Anders ausgedrückt, gibt es in einer FullHouse-Einheit genau eine leere Zelle.

- Ein **NakedSingle** ist eine Zahl, die in eine eindeutige Zelle gesetzt werden muss, weil alle anderen Zahlen für diese Zelle ausgeschlossen worden sind. Diese eindeutige Zelle nennen wir eine **NakedSingle-Zelle**.

  ▷ Beispiel: In Abbildung 12 ist die Zelle $z_{9,9}$ eine NakedSingle-Zelle. Für diese Zelle sind aufgrund des sie enthaltenden Blocks bereits die Zahlen 8, 9 ausgeschlossen worden. Wegen der sie enthaltenden Spalte sind 4, 5, 6 und wegen der sie enthaltenden Zeile sind 1, 2 und 3 ausgeschlossen worden. Die 7 ist dann das NakedSingle.

  ▷ Hinweis: Wenn Sie für jede Zelle eine Liste der Zahlen anlegen, die noch für diese Zelle in Frage kommen, so ist eine NakedSingle-Zelle eine Zelle, deren Liste nur noch aus einer Zahl besteht, dem NakedSingle.

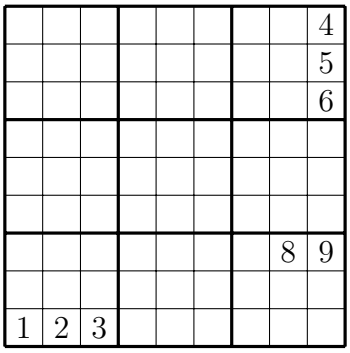

Abbildung 12: Zelle $z_{9,9}$ ist eine NakedSingle-Zelle.

- Ein **HiddenSingle** ist eine Zahl, die nur noch in eine eindeutige Zelle gesetzt werden kann, weil alle anderen Zellen für diese Zahl ausgeschlossen worden sind. Diese eindeutige Zelle nennen wir eine **HiddenSingle-Zelle**.

▷ Hinweis: Die Liste der Zahlen, die in eine HiddenSingle-Zelle geschrieben werden können, kann noch „recht lang“ sein. Betrachten Sie beispielsweise eine Zeile, in der es nur noch eine Liste gibt, welche die Zahl 1 enthält. Dann ist die 1 ein HiddenSingle in dieser Zeile.

▷ Beispiel: In Abbildung 13 ist $z_{9,9}$ eine HiddenSingle-Zelle. Für die Zahl 1, das HiddenSingle, können alle anderen Zellen in dem Block (sogar im ganzen Gitter) ausgeschlossen werden.

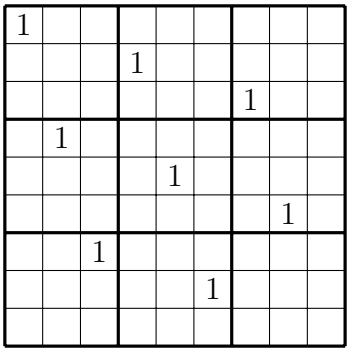


Abbildung 13: Zelle $z_{9,9}$ ist eine HiddenSingle-Zelle.

- Ein **NakedPair** ist ein Paar von Zahlen, die in zwei Zellen gesetzt werden müssen, weil alle anderen Zahlen für diese Zellen ausgeschlossen worden sind. Diese zwei Zellen nennen wir ein **NakedPair-Zellpaar**. Keine der Zellen aus diesem NakedPair-Zellpaar darf dabei eine NakedSingle- oder HiddenSingle-Zelle sein.

  ▷ Anders ausgedrückt: Es ist zwar für das Paar der Zellen klar, dass dort nur zwei Zahlen hineingehören, aber es ist nicht klar, welche der beiden Zahlen in welche der beiden Zellen geschrieben werden muss.

  ▷ Beispiel: In Abbildung 14 bilden die Zellen $z_{8,1}, z_{9,1}$ ein NakedPair-Zellpaar. Die Zahlen 7 und 8 bilden dabei das NakedPair.

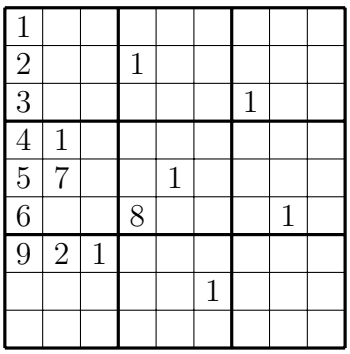


Abbildung 14: Die Zellen $z_{8,1}$ und die Zellen $z_{9,1}$ bilden ein NakedPair-Zellpaar, wobei die Zahlen 7 und 8 das NakedPair darstellen. Diese beiden Zahlen sind für die angegebenen Zellen die einzig noch übrig gebliebenen Kandidaten.

- Ein **HiddenPair** ist ein Paar von Zahlen, die in zwei Zellen gesetzt werden müssen, weil alle anderen Zellen für diese Zahlen ausgeschlossen worden sind. Diese Zellen nennen wir **HiddenPair-Zellpaar**. Keine der Zellen aus diesem HiddenPair-Zellpaar darf dabei eine NakedSingle- oder HiddenSingle-Zelle sein.

▷ Anders ausgedrückt: Es ist zwar für das Paar der Zellen klar, dass dort nur zwei Zahlen hineingehören, aber es ist nicht klar, welche der beiden Zahlen in welche der beiden Zellen geschrieben werden muss.

▷ Beispiel: Siehe Abbildung 15.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | 1 | | | | | |
| | | | | | | 1 | | 9 |
| | 1 | | | | | | | |
| | | | | 1 | | | | |
| | | | | | | | 1 | 8 |
| | 9 | 1 | | 8 | | | | |
| | 8 | | | 9 | 1 | | | |
| 7 | 6 | 5 | | | | | | |

Abbildung 15: Die Zellen $z_{9,7}$ und $z_{9,8}$ bilden ein HiddenPair-Zellpaar, wobei die 8 und die 9 das HiddenPair bilden. Alle anderen Zellen in dem unteren, leeren Block sind für 8 und 9 ausgeschlossen, so dass nur noch die beiden angegebenen Zellen für 8 und 9 in Frage kommen.

# 3 Erläuterungen zu den Methoden der Interfaces

In den vorgegebenen Methoden können Sie das Wort `<Unit>` mit dem Wort `Block`, `Row` oder `Col` ersetzen. Ein `<Unit>` entspricht einer „Einheit“ (siehe Kap. 2).

Sollten Begriffe, die im Folgenden auftauchen, nicht in den Definitionen stehen, so melden Sie sich bitte bei uns. Dann werden wir zeitnah eine aktualisierte Version dieses Arbeitsblattes vorgeben. Auch andere sachdienliche Hinweise sind willkommen.

Methodenversionen ohne `<Unit>` und/oder `anchor` gelten für das gesamte Gitter.
Sie haben noch nicht mit Schnittstellen gearbeitet? Versuchen Sie es doch mal mit einschlägiger Fachliteratur! Ein Standardwerk ist beispielsweise „Java ist auch eine Insel“. Der Rheinwerk-Verlag offeriert sogar eine kostenfreie online-Version:
`http://openbook.rheinwerk-verlag.de/javainsel/`.
Ein weiterer Vorschlag:
`https://docs.oracle.com/javase/tutorial/java/IandI/index.html`.

Wenn eine Zellenmenge **aufsteigend sortiert** wird, ist damit gemeint, dass an der ersten Stelle die kleinste Zelle steht, an der zweiten Stelle die zweit-kleinste Zelle steht, und so weiter. Da es in der Programmierkursklausur mit dem Begriff **aufsteigend sortiert** Verständnisprobleme gab, erwähne ich diesen Begriff an dieser Stelle nochmal gesondert.

## 3.1 SolvingUtil

### 3.1.1 Full Houses and White Spaces

(1) `List<Cell> getWhiteSpaces(Grid grid)`:

- Gibt aufsteigend sortiert alle Zellen von `grid` zurück, welche eine `-1` beinhalten. Gibt es keine leeren Zellen, so ist die Liste dementsprechend leer.

(2) `List<Cell> get<Unit>WhiteSpaces(Grid grid, Cell anchor)`:

- Gibt aufsteigend sortiert alle Zellen der Einheit zurück, welche eine `-1` beinhalten. Die Einheit wird durch `anchor` eindeutig festgelegt.Enthält die Einheit keine leeren Zellen, so ist die Liste dementsprechend leer.

(3) `boolean hasFullHouseUnit(Grid grid)`:

- Entscheidet, ob in `grid` eine FullHouse-Einheit existiert; liefert den Wert `true`, falls ja, `false` sonst.

(4) `boolean isFullHouse<Unit>(Grid grid, Cell anchor)`:

- Entscheidet, ob die Einheit eine FullHouse-Einheit ist; liefert den Wert `true`, falls ja, `false` sonst. Die Einheit wird durch `anchor` eindeutig festgelegt.

### 3.1.2 Singles

(1) `Cell getMinimalHiddenSingleCell(Grid grid)`:

- Analog zu `get<Unit>MinimalHiddenSingleCell` (siehe (2)). Bezieht sich hier statt auf die Einheit auf das gesamte Grid.

(2) `Cell get<Unit>MinimalHiddenSingleCell(Grid grid, Cell anchor)`:

- Analog zur NakedSingle-Version (siehe (4)).

(3) `Cell getMinimalNakedSingleCell(Grid grid)`:

- Analog zu `get<Unit>MinimalNakedSingleCell` (siehe (4)).

(4) `Cell get<Unit>MinimalNakedSingleCell(Grid grid, Cell anchor)`:

- Gibt die minimale NakedSingle-Zelle der Einheit zurück. Die Einheit wird durch `anchor` eindeutig festgelegt. Dabei brauchen Sie nur den Fall zu betrachten, in dem die Einheit mindestens eine NakedSingle-Zelle enthält. Andere Fälle müssen Sie mit dieser Methode nicht abfangen.

(5) `boolean hasNakedSingleCell(Grid grid)`:

- Analog zu `is<Unit>WithNakedSingleCell` (siehe (8)).

(6) `boolean hasHiddenSingleCell(Grid grid)`;

- Analog zu `is<Unit>WithHiddenSingleCell` (siehe (7)).

(7) `boolean is<Unit>WithHiddenSingleCell(Grid grid, Cell anchor)`:

- Analog zur NakedSingle-Version (siehe (8)).

(8) `boolean is<Unit>WithNakedSingleCell(Grid grid, Cell anchor)`:

- Entscheidet, ob die Einheit mindestens eine NakedSingle-Zelle enthält. Die Einheit wird durch `anchor` eindeutig festgelegt.

### 3.1.3 Pairs

(1) `Cell[] getMinimalHiddenPairCells(Grid grid)`:

- Analog zur NakedPair-Version (siehe (3)).

(2) `Cell[] get<Unit>MinimalHiddenPairCells(Grid grid, Cell anchor)`:

- Analog zur NakedPair-Version (siehe (4)).

(3) `Cell[] getMinimalNakedPairCells(Grid grid)`:

- Gibt das minimale NakedPair-Zellpaar von `grid` zurück. Dabei sind die zwei in `Cell[]` gespeicherten Zellen aufsteigend sortiert. Gibt es kein NakedPair-Zellpaar, so ist dementsprechend `Cell[]` leer.

(4) `Cell[] get<Unit>MinimalNakedPairCells(Grid grid, Cell anchor)`:

- Gibt das minimale NakedPair-Zellpaar der Einheit zurück. Dabei sind die zwei in `Cell[]` gespeicherten Zellen aufsteigend sortiert. Gibt es kein NakedPair-Zellpaar in der Einheit, so ist dementsprechend `Cell[]` leer.
  Die Einheit wird durch `anchor` eindeutig festgelegt.

(5) `boolean is<Unit>WithHiddenPairCells(Grid grid, Cell anchor)`:

- Analog zur NakedPair-Version (siehe (6)).

(6) `boolean is<Unit>WithNakedPairCells(Grid grid, Cell anchor)`:

- Entscheidet, ob die Einheit ein NakedPair-Zellpaar enthält; liefert den Wert `true`, falls ja, sonst `false`. Die Einheit wird durch `anchor` eindeutig festgelegt.

(7) `boolean hasHiddenPairCells(Grid grid)`:

- Entscheidet, ob `grid` ein HiddenPair-Zellpaar enthält; liefert den Wert `true`, falls ja, sonst `false`.

(8) `boolean hasNakedPairCells(Grid grid)`:

- Entscheidet, ob `grid` ein NakedPair-Zellpaar enthält; liefert den Wert `true`, falls ja, sonst `false`.

### 3.1.4 Validity and Solving

(1) `boolean isValid(Grid grid)`:

- Entscheidet, ob in `grid` eine zulässige Belegung vorliegt; liefert den Wert `true`, falls ja, `false` sonst.

(2) `boolean isValid<Unit>(Grid grid, Cell anchor)`:

- Entscheidet, ob in der Einheit eine zulässige Belegung vorliegt; liefert den Wert `true`, falls ja, `false` sonst. Die Einheit wird durch `anchor` eindeutig festgelegt.

(3) `List<Grid> solve<Unit>Based(Grid grid)`

- Siehe **??**.

## 3.2 IsoUtil

### 3.2.1 Blockinterne Zeilen- und Spaltenpermutationen

(1) `int[] getBlockInternColPermutationImage(Grid grid1, Grid grid2, Cell anchor);`

- Gibt den Bildvektor zurück, der die blockinterne Spaltenpermutation beschreibt, welche auf den Block, festgelegt durch `anchor`, in `grid1` angewendet wurde, um `grid2` zu erhalten. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine blockinterne Spaltenpermutation von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

(2) `int[] getBlockInternRowPermutationImage(Grid grid1, Grid grid2, Cell anchor);`

- Gibt den Bildvektor zurück, der die blockinterne Zeilenpermutation beschreibt, welche auf den Block, festgelegt durch `anchor`, in `grid1` angewendet wurde, um `grid2` zu erhalten. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine blockinterne Zeilenpermutation von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

(3) `boolean isBlockInternColPermutation(Grid grid1, Grid grid2);`

- Entscheidet, ob `grid2` aus `grid1` entstanden ist, indem eine blockinterne Spaltenpermutation auf `grid1` angewendet wurde; falls ja, wird `true` zurückgeliefert, sonst `false`.

(4) `boolean isBlockInternRowPermutation(Grid grid1, Grid grid2);`

- Entscheidet, ob `grid2` aus `grid1` entstanden ist, indem eine blockinterne Zeilenpermutation auf `grid1` angewendet wurde; falls ja, wird `true` zurückgeliefert, sonst `false`.

(5) `void applyBlockInternColPermutation(Grid grid, Cell anchor, int[] image)`:

- Analog zu `applyBlockInternRowPermutation` (siehe (6)).

(6) `void applyBlockInternRowPermutation(Grid grid, Cell anchor, int[] image)`:

- Wendet die durch den Bildvektor `image` beschriebene blockinterne Zeilenpermutation auf den Block an, dessen Ankerzelle in `anchor` abzulesen ist.

### 3.2.2 Blockpermutationen

(1) `Cell[] getBlockPermutationImage(Grid grid1, Grid grid2)`:

- Gibt den Blockbildvektor zurück, der die Blockpermutation beschreibt, welche auf `grid1` angewendet wurde, so dass `grid2` entstanden ist. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine Blockpermutation von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

(2) `boolean isBlockPermutation(Grid grid1, Grid grid2)`:

- Entscheidet, ob `grid2` aus `grid1` entstanden ist, indem eine Blockpermutation vorgenommen wurde; falls ja, wird `true` zurückgeliefert, sonst `false`.

(3) `void applyBlockPermutation(Grid grid, Cell[] image)`:

- Wendet die durch `image` beschriebene Blockpermutation auf `grid` an.

### 3.2.3 Wertpermutationen

(1) `boolean is<Unit>ValuePermutation(Grid grid1, Grid grid2, Cell anchor)`:

- Entscheidet, ob die Einheit in `grid2` durch eine Wertpermutation der Einheit in `grid1` entstanden ist. Falls ja, wird `true` geliefert, sonst `false`. Beide Einheiten sind durch `anchor` jeweils eindeutig festgelegt.

(2) `void apply<Unit>ValuePermutation(Grid grid, Cell anchor, int[] image)`:

- Wendet die durch den Bildvektor `image` beschriebene Wertpermutation auf die Einheit in `grid` an. Die Einheit wird durch `anchor` eindeutig festgelegt.

(3) `int[] get<Unit>ValuePermutationImage(Grid grid1, Grid grid2, Cell anchor)`:

- Gibt den Bildvektor der Wertpermutation wieder, welche die Einheit von `grid1` in die Einheit von `grid2` überführt hat. Die Einheit wird durch `anchor` eindeutig festgelegt. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine Wertpermutation von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

## 3.3 GridUtil1, 2 und 3

(1) `Cell getReflectionAnchor(Grid grid1, Grid grid2)`:

- Gibt die Ankerzelle der Spalte bzw. Zeile zurück, an der `grid1` gespiegelt wurde, so dass `grid2` erzeugt wurde. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine Spiegelung von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

(2) `boolean isReflection(Grid grid1, Grid grid2)`:

- Entscheidet, ob `grid2` durch eine Spiegelung von `grid1` entstanden ist.

(3) `boolean isTransposition(Grid grid1, Grid grid2)`:

- Entscheidet, ob `grid2` durch Transposition von `grid1` entstanden ist; falls ja, wird `true` zurückgeliefert, ansonsten `false`.

(4) `void reflect(Grid grid, Cell anchor)`:

- Spiegelt `grid` an der durch `anchor` festgelegten Zeile beziehungsweise Spalte.

(5) `void transponse(Grid grid)`:

- Transponiert das Gitter `grid`.

(6) `void turnRight(Grid grid)`:

- Führt eine Rechtsdrehung auf `grid` aus.

(7) `boolean isGridTurn(Grid grid1, Grid grid2)`:

- Entscheidet, ob `grid2` durch Rechtsdrehungen von `grid1` entstanden ist; falls ja, wird `true` zurückgegeben, sonst `false`. Beachten Sie, dass die Identität (also das Nichtverändern des Gitters) als vierfache Rechtsdrehung anzusehen ist.

(8) `int getGridTurnNumber(Grid grid1, Grid grid2)`:

- Gibt die Anzahl an Rechtsdrehungen wieder, die auf `grid1` angewendet werden mussten, um `grid2` zu bekommen. Zulässige Rückgabewerte liegen zwischen 0 und 3. Dabei brauchen Sie nur den Fall zu betrachten, in dem `grid2` durch eine feste Anzahl an Rechtsdrehungen von `grid1` entstanden ist. Andere Fälle müssen Sie mit der Methode nicht abfangen.

## 3.4 <Unit>BasedSolvers - reboot

- Initialisiere Kandidaten-Array:

  ▷ Mit *Kandidaten-Array* bezeichnen wir ein 9 x 9 Array. Dieses legen Sie am besten global in Ihrer Klasse an. Jede Zelle des Kandidaten-Arrays steht für eine Zelle in dem vorgegebenen Grid. Der Einfachheit halber bezeichnen wir die Zelle im Kandidaten-Array, die für $z_{i,j}$ steht, mit $k_{i,j}$. Jede Zelle $k_{i,j}$ beinhaltet wiederum ein `int`-Array der Länge 9 (oder eine Liste oder sonst irgendetwas in der Art), welche wir mit $L_{i,j}$ bezeichnen. In $L_{i,j}$ stehen alle die Zahlen, die als Kandidaten für die Zelle in Frage kommen, und gar keine Zahlen, wenn bereits eine Zahl in die Zelle eingetragen wurde. Steht also beispielsweise in $z_{1,2}$ bereits die Zahl 4, so enthält $L_{1,2}$ gar keine Zahlen. Steht in $z_{1,2}$ noch keine Zahl, ist aber nur noch die 4 für diese Zelle möglich, weil beispielsweise Zeile 1 eine Fullhouse-Einheit ist, so enthält $L_{1,2}$ nur noch die Zahl 4. Die Kandidatenlisten passen Sie also so an, dass für die Zellen, in denen bereits eine Zahl drin steht, die Kandidatenlisten geleert werden. Weiterhin passen Sie die Kandidatenlisten so an, dass Sie, wenn in Zelle $z_{1,2}$ die Zahl 4 drin steht, einmal im Block mit Ankerzelle $z_{1,1}$, einmal in Zeile 1 und einmal in Spalte 2 aus allen Kandidatenlisten die Zahl 4 raus streichen. Sind nun diese Anpassungen vorgenommen, begeben wir uns in den Algorithmus, also zu Schritt 1.

- Update:

  ▷ Mit **Update** ist gemeint, dass Sie die Zahlen, die für eine Zelle in Frage kommen, für alle Zellen aktualisieren, nachdem Sie entweder eine neue Zahl in eine Zelle geschrieben oder ein HiddenPair beziehungsweise ein NakedPair gefunden haben.
  Haben Sie also beispielsweise in Schritt 3 für $z_{5,4}$ ein NakedSingle gefunden, zum Beispiel die Zahl 7, tragen Sie die 7 in $z_{5,4}$ ein („Belege in dieser Einheit...“). Die Zahlen in der Liste $L_{5,4}$ werden gelöscht. Weiterhin streichen Sie aus allen Listen in Zeile 5, Spalte 4 und Block mit Ankerzelle $z_{4,4}$ die Zahl 7 raus.
  Haben Sie beispielsweise in Schritt 5 das NakedPair 1 und 2 in Zeile 7 gefunden, sagen wir in Zelle $z_{7,1}$ und $z_{7,9}$, passen Sie die Kandidatenlisten $L_{7,1}$ und $L_{7,9}$ an. Dann macht die Update-Prozedur noch Folgendes: In allen Listen werden die Zahlen 1 und 2 gestrichen. Beachten Sie dabei: Tatsächlich wird in diesem Fall nichts in das Grid geschrieben, sondern nur die Kandidatenlisten aktualisiert.

Die Arbeitsweise der Methode `List<Grid> solve<Unit>Based(Grid grid)` lässt sich in Pseudocode folgendermaßen beschreiben.

---

Schritt 0: Initialisiere Kandidaten-Array.
Schritt 1: Ist das Sudoku vollständig und zulässig gefüllt? (Kontrollieren Sie das <Unit>-weise.)
    Falls ja: Stop.
    Falls nein: Gehe zu Schritt 2.
Schritt 2: Gibt es (mind.) eine Einheit, die eine Fullhouse-Einheit ist?
    Falls ja:
      Wähle Einheit mit minimaler Ankerzelle.

Belege in dieser Einheit die einzig verbleibende Zelle mit der einzig möglichen Zahl.
Update.
Gehe zu Schritt 1.
Falls nein: Gehe zu Schritt 3.

Schritt 3: Gibt es (mind.) eine Einheit, die mindestens eine NakedSingle-Zelle hat?
Falls ja:
Wähle Einheit mit minimaler Ankerzelle.
Belege in dieser Einheit die minimale Zelle mit der einzig möglichen Zahl.
Update.
Gehe zu Schritt 1.
Falls nein: Gehe zu Schritt 4.

Schritt 4: Gibt es (mind.) eine Einheit mit einer HiddenSingle-Zelle?
Falls ja:
Wähle Einheit mit minimaler Ankerzelle.
Belege in dieser Einheit die minimale Zelle mit der einzig möglichen Zahl.
Update.
Gehe zu Schritt 1.
Falls nein: Gehe zu Schritt 5.

Schritt 5: Gibt es eine Einheit mit einem NakedPair-Zellpaar?
Falls ja:
Wähle die Einheit mit minimaler Ankerzelle und das minimale Zellpaar.
Update entsprechend minimalem Zellpaar.
Falls Update in mindestens einer Kandidatenliste etwas geändert hat, gehe zu Schritt 1.
Falls nein: Gehe zu Schritt 6.

Schritt 6: Gibt es eine Einheit mit einem HiddenPair-Zellpaar?
Falls ja:
Wähle die Einheit mit minimaler Ankerzelle und das minimale Zellpaar.
Update entsprechend minimalem Zellpaar.
Falls Update in mindestens einer Kandidatenliste etwas geändert hat, gehe zu Schritt 1.
Falls nein: Stop.

---

Zurückgegeben werden soll eine Liste mit Grids (`List<Grid>`). In dieser Liste ist an erster Stelle das Ausgangsgitter gespeichert. Sobald im Laufe des obigen Algorithmus eine Zahl in das Gitter eingetragen wurde, wird an die nächste freie Stelle der Liste das neue, mit genau einer weiteren Zahl gefüllte Gitter gespeichert.
Hinweis: Es **kann** sein, dass das letzte einzutragende Sudoku noch immer nicht vollständig gelöst ist. Das ist durch die Vorgaben auch nicht verlangt, ist also, wenn es mal vorkommt, völlig in Ordnung, so lange alle anderen Vorgaben erfüllt sind.

Aufgrund der besseren Lesbarkeit starten wir jetzt eine neue Seite...

# 4 Interfaces

## 4.1 Interface BlockIsoUtil

```
interface BlockIsoUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um mit zueinander isomorphen Sudokus zu arbeiten.
    * Methoden beziehen sich auf die Einheit 'Block'.
    */

    //BlockPermutation
    void applyBlockPermutation(Grid grid, Cell[] image);
    boolean isBlockPermutation(Grid grid1, Grid grid2);
    Cell[] getBlockPermutationImage(Grid grid1, Grid grid2);

    //ValuePermutation
    void applyBlockValuePermutation(Grid grid, Cell anchor, int[] image);
    int[] getBlockValuePermutationImage(Grid grid1, Grid grid2, Cell anchor);
    // Achtung: Diese Methode ist neu! (2018 05 03)
    boolean isBlockValuePermutation(Grid grid1, Grid grid2, Cell anchor);
}
```

## 4.2 Interface RowIsoUtil

```
interface RowIsoUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um mit zueinander isomorphen Sudokus zu arbeiten.
    * Methoden beziehen sich auf die Einheit 'Row' (also Zeile).
    */

    //BlockInternRow
    void applyBlockInternRowPermutation(Grid grid, Cell anchor, int[] image);
    boolean isBlockInternRowPermutation(Grid grid1, Grid grid2);
    int[] getBlockInternRowPermutationImage(Grid grid1, Grid grid2, Cell anchor);

    //ValuePermutation
    void applyRowValuePermutation(Grid grid, Cell anchor, int[] image);
    int[] getRowValuePermutationImage(Grid grid1, Grid grid2, Cell anchor);
    // Achtung: Diese Methode ist neu! (2018 05 03)
    boolean isRowValuePermutation(Grid grid1, Grid grid2, Cell anchor);

}
```

## 4.3 Interface ColIsoUtil

```
interface ColIsoUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um mit zueinander isomorphen Sudokus zu arbeiten.
    * Methoden beziehen sich auf die Einheit 'Col' (also Spalte).
    */

    //BlockInternCol
    void applyBlockInternColPermutation(Grid grid, Cell anchor, int[] image);
    boolean isBlockInternColPermutation(Grid grid1, Grid grid2);
    int[] getBlockInternColPermutationImage(Grid grid1, Grid grid2, Cell anchor);

    //ValuePermutation
    void applyColValuePermutation(Grid grid, Cell anchor, int[] image);
    int[] getColValuePermutationImage(Grid grid1, Grid grid2, Cell anchor);
    // Achtung: Diese Methode ist neu! (2018 05 03)
    boolean isColValuePermutation(Grid grid1, Grid grid2, Cell anchor);

}
```

## 4.4 Interface BlockSolvingUtil

```
interface BlockSolvingUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um Sudokus loesen zu koennen.
    * Die Methoden beziehen sich auf die Einheit 'Block'.
    */

    //Feasibility and Whitespaces
    boolean isValidBlock(Grid grid, Cell anchor);
    List<Cell> getBlockWhiteSpaces(Grid grid, Cell anchor);

    //Singles
    //FullHouse Singles
    boolean hasFullHouseBlock(Grid grid);
    boolean isFullHouseBlock(Grid grid, Cell anchor);

    //NakedSingles
    boolean isBlockWithNakedSingleCell(Grid grid, Cell anchor);
    Cell getBlockMinimalNakedSingleCell(Grid grid, Cell anchor);

    //HiddenSingles
    boolean isBlockWithHiddenSingleCell(Grid grid, Cell anchor);
    Cell getBlockMinimalHiddenSingleCell(Grid grid, Cell anchor);

    //Pairs
    //Naked Pairs
    boolean isBlockWithNakedPair(Grid grid, Cell anchor);
    Cell[] getBlockNakedPairCells(Grid grid, Cell anchor);

    //HiddenPairs
    boolean isBlockWithHiddenPair(Grid grid, Cell anchor);
    Cell[] getBlockHiddenPairCells(Grid grid, Cell anchor);

    //BlockBasedSolver
    List<Grid> solveBlockBased(Grid grid);

}
```

## 4.5 Interface ColSolvingUtil

```
interface ColSolvingUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um Sudokus loesen zu koennen.
    * Die Methoden beziehen sich auf die Einheit 'Col'.
    */

    //Feasibility and Whitespaces
    boolean isValidCol(Grid grid, Cell anchor);
    List<Cell> getColWhiteSpaces(Grid grid, Cell anchor);

    //Singles
    //FullHouse Singles
    boolean hasFullHouseCol(Grid grid);
    boolean isFullHouseCol(Grid grid, Cell anchor);

    //NakedSingles
    boolean isColWithNakedSingleCell(Grid grid, Cell anchor);
    Cell getColMinimalNakedSingleCell(Grid grid, Cell anchor);

    //HiddenSingles
    boolean isColWithHiddenSingleCell(Grid grid, Cell anchor);
    Cell getColMinimalHiddenSingleCell(Grid grid, Cell anchor);

    //Pairs
    //Naked Pairs
    boolean isColWithNakedPairCells(Grid grid, Cell anchor);
    Cell[] getColMinimalNakedPairCells(Grid grid, Cell anchor);

    //HiddenPairs
    boolean isColWithHiddenPairCells(Grid grid, Cell anchor);
    Cell[] getColMinimalHiddenPairCells(Grid grid, Cell anchor);

    //ColBasedSolver
    List<Grid> solveColBased(Grid grid);


}
```

## 4.6 Interface RowSolvingUtil

```
interface RowSolvingUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um Sudokus loesen zu koennen.
    * Die Methoden beziehen sich auf die Einheit 'Row'.
    */

    //Feasibility and Whitespaces
    boolean isValidRow(Grid grid, Cell anchor);
    List<Cell> getRowWhiteSpaces(Grid grid, Cell anchor);

    //Singles
    //FullHouse Singles
    boolean hasFullHouseRow(Grid grid);
    boolean isFullHouseRow(Grid grid, Cell anchor);

    //NakedSingles
    boolean isRowWithNakedSingleCell(Grid grid, Cell anchor);
    Cell getRowMinimalNakedSingleCell(Grid grid, Cell anchor);

    //HiddenSingles
    boolean isRowWithHiddenSingleCell(Grid grid, Cell anchor);
    Cell getRowMinimalHiddenSingleCell(Grid grid, Cell anchor);

    //Pairs
    //Naked Pairs
    boolean isRowWithNakedPairCells(Grid grid, Cell anchor);
    Cell[] getRowMinimalNakedPairCells(Grid grid, Cell anchor);

    //HiddenPairs
    boolean isRowWithHiddenPairCells(Grid grid, Cell anchor);
    Cell[] getRowMinimalHiddenPairCells(Grid grid, Cell anchor);

    //RowBasedSolver
    List<Grid> solveRowBased(Grid grid);


}
```

## 4.7 Interface GridUtil1

```
interface GridUtil1{
    /**
    * Beinhaltet Methoden bezueglich der Transposition
    * von Gittern.
    */

    //GridTransposition
    void transponse(Grid grid);
    boolean isTransposition(Grid grid1, Grid grid2);
}
```

## 4.8 Interface GridUtil2

```
interface GridUtil2{
    /**
    * Beinhaltet Methoden bezueglich der Spiegelung
    * von Gittern.
    */

    //GridReflection
    void reflect(Grid grid, Cell anchor);
    boolean isReflection(Grid grid1, Grid grid2);
    Cell getReflectionAnchor(Grid grid1, Grid grid2);
}
```

## 4.9 Interface GridUtil3

```
interface GridUtil3{
    /**
    * Beinhaltet Methoden bezueglich der Drehung
    * von Gittern.
    */

    //GridRightTurns
    void turnRight(Grid grid);
    boolean isGridTurn(Grid grid1, Grid grid2);
    int getGridTurnNumber(Grid grid1, Grid grid2);
}
```

## 4.10 Interface GridSolvingUtil

```
interface GridSolvingUtil {
    /**
    * Beinhaltet hilfreiche Methoden,
    * um Sudokus loesen zu koennen.
    */

    //Feasibility and Whitespaces
    boolean isValid(Grid grid);
    List<Cell> getWhiteSpaces(Grid grid);

    //Singles
    //FullHouse Singles
    boolean hasFullHouseUnit(Grid grid);

    //NakedSingles
    boolean hasNakedSingleCell(Grid grid);
    Cell getMinimalNakedSingleCell(Grid grid);

    //HiddenSingles
    boolean hasHiddenSingleCell(Grid grid1, Cell anchor);
    Cell getMinimalHiddenSingleCell(Grid grid);

    //Pairs
    //Naked Pairs
    boolean hasNakedPair(Grid grid);
    Cell[] getMinimalNakedPairCells(Grid grid);

    //HiddenPairs
    boolean hasHiddenPairCells(Grid grid);
    Cell[] getMinimalHiddenPairCells(Grid grid);

}
```

# 5 Aufgaben

- Hinweise zur Bewertung: Wie bereits in der Vorbesprechung angegeben, müssen Sie für das Portfolio insgesamt 150 Punkte erreichen, damit es zumindest als „bestanden“ gilt. Unter Umständen ist es notwendig, von den 150 Punkten nach unten hin abzuweichen. Seien Sie sich aber bitte bewusst, dass Sie **in jedem Fall insgesamt mindestens 50 Punkte in den Programmieraufgaben erreichen müssen**, um das Portfolio zu bestehen.

- In den Aufgaben wird auf die **Javadoc-Aufgabe** verwiesen. Diese lautet wie folgt: Erstellen Sie **in Ihrem Code** Javadoc-taugliche Kommentare zu all Ihren Methoden. Verwenden Sie Tags. Bei der Kommentierung Ihres Codes ist die Verwendung von `@author <Vorname, Nachname>` verpflichtend. Gibt es für eine Klasse nur einen Autor, dann reicht es, den `author`-Tag über die Klasse zu schreiben. Besitzt eine Klasse mehrere Autoren, dann ist für jeden Autor über der Klasse ein Tag zu setzen und über jede Methode. Allerdings darf es für eine Methode nur genau einen Autor geben. Des Weiteren muss Ihr (aus den Kommentaren generierte) Javadoc auch entsprechend Informationen zu den Autoren enthalten. Daher sollte das `javadoc`-Tool mit der Option `-author` aufgerufen werden.

  Ihre Dokumentation darf entweder ganz in Deutsch oder ganz in Englisch verfasst werden. Wir empfehlen Englisch.

  Sie dürfen die Definitionen aus Kapitel 2 als bekannt voraussetzen. Sie müssen die Begriffe also nicht nochmals in den Kommentaren erläutern.

- Wir wünschen Ihnen viel Erfolg!

## 5.1 Gruppenaufgabe — 50 Punkte

(a) **Programmieraufgabe** — **30 Punkte**
Schreiben Sie eine Klasse namens `GridSolvingUtils`, welche das Interface `GridSolvingUtil` implementiert.

(b) **Theorieaufgabe** — **20 Punkte**
Siehe **Javadoc-Aufgabe**.

## 5.2 Individualaufgaben Typ „Block“ 100 Punkte

(a) **Programmieraufgabe** **50 Punkte**
Schreiben Sie eine Klasse namens `BlockUtils`, welche das Interface `BlockIsoUtil` sowie das Interface `BlockSolvingUtil` implementiert.

Schreiben Sie ein weitere Klasse namens `GridUtils1`, welche das Interface `GridUtil1` implementiert.

(b) **Theorieaufgabe** **20 Punkte**
Siehe **Javadoc-Aufgabe**.

(c) **Theorieaufgabe** **15 Punkte**
Weisen Sie nach, dass ein zulässig belegtes Sudoku durch eine Transposition noch immer ein zulässig belegtes Sudoku ist.

(d) **Theorieaufgabe** **15 Punkte**
Nehmen Sie an, es sei Ihre Aufgabe, eine Methode zu schreiben, die entscheidet, ob zwei Sudokus durch eine Verkettung der Abbildungen

(1) Wertpermutation auf einem Sudoku
(2) blockinterne Spaltenpermutation
(3) blockinterne Zeilenpermutation
(4) horizontale Blockpermutation
(5) vertikale Blockpermutation
(6) Transposition
(7) Spiegelung
(8) Rechtsdrehung

entstanden ist.

Wie würden Sie diese Aufgabe umsetzen?

Skizzieren Sie die Methoden, die Sie zusätzlich schreiben würden (Name und Beschreibung, wie in Kapitel 3, kein detaillierter Code).

Geben Sie, zum Beispiel in Pseudocode an, wie Sie Ihre Methoden miteinander verknüpfen würden.

Wir werden Ihren Vorschlag nach folgenden Kriterien beurteilen:

- Terminiert Ihre Methode?
- Arbeitet die Methode korrekt?
- Arbeitet Ihre Methode (zumindest in der Theorie) effizient, also vermeidet sie beispielsweise unnötige Schritte?

Sie dürfen Ihren Vorschlag als Teamvorschlag abgeben. Das heißt, Sie dürfen innerhalb Ihrer Gruppe zusammenarbeiten und eine gemeinsame Lösung abgeben.

## 5.3 Individualaufgaben Typ „Col“ 100 Punkte

(a) **Programmieraufgabe** **50 Punkte**
Schreiben Sie eine Klasse namens `ColUtils`, welche das Interface `ColIsoUtil` sowie das Interface `ColSolvingUtil` implementiert.

Schreiben Sie ein weitere Klasse namens `GridUtils2`, welche das Interface `GridUtil2` implementiert.

(b) **Theorieaufgabe** **20 Punkte**
Siehe **Javadoc-Aufgabe**.

(c) **Theorieaufgabe** **15 Punkte**
Weisen Sie nach, dass ein zulässig belegtes Sudoku durch eine Spiegelung noch immer ein zulässig belegtes Sudoku ist.

(d) **Theorieaufgabe** **15 Punkte**
Nehmen Sie an, es sei Ihre Aufgabe, eine Methode zu schreiben, die entscheidet, ob zwei Sudokus durch eine Verkettung der Abbildungen

(1) Wertpermutation auf einem Sudoku
(2) blockinterne Spaltenpermutation
(3) blockinterne Zeilenpermutation
(4) horizontale Blockpermutation
(5) vertikale Blockpermutation
(6) Transposition
(7) Spiegelung
(8) Rechtsdrehung

entstanden ist.

Wie würden Sie diese Aufgabe umsetzen?

Skizzieren Sie die Methoden, die Sie zusätzlich schreiben würden (Name und Beschreibung, wie in Kapitel 3, kein detaillierter Code).

Geben Sie, zum Beispiel in Pseudocode an, wie Sie Ihre Methoden miteinander verknüpfen würden.

Wir werden Ihren Vorschlag nach folgenden Kriterien beurteilen:

- Terminiert Ihre Methode?
- Arbeitet die Methode korrekt?
- Arbeitet Ihre Methode (zumindest in der Theorie) effizient, also vermeidet sie beispielsweise unnötige Schritte?

Sie dürfen Ihren Vorschlag als Teamvorschlag abgeben. Das heißt, Sie dürfen innerhalb Ihrer Gruppe zusammenarbeiten und eine gemeinsame Lösung abgeben.

## 5.4 Individualaufgaben Typ „Row“ 100 Punkte

(a) **Programmieraufgabe** **50 Punkte**
Schreiben Sie eine Klasse namens `RowUtils`, welche das Interface `RowIsoUtil` sowie das Interface `RowSolvingUtil` implementiert.

Schreiben Sie ein weitere Klasse namens `GridUtils3`, welche das Interface `GridUtil3` implementiert.

(b) **Theorieaufgabe** **20 Punkte**
Siehe **Javadoc-Aufgabe**.

(c) **Theorieaufgabe** **15 Punkte**
Weisen Sie nach, dass ein zulässig belegtes Sudoku durch eine Rechtsdrehung noch immer ein zulässig belegtes Sudoku ist.

(d) **Theorieaufgabe** **15 Punkte**
Nehmen Sie an, es sei Ihre Aufgabe, eine Methode zu schreiben, die entscheidet, ob zwei Sudokus durch eine Verkettung der Abbildungen

(1) Wertpermutation auf einem Sudoku
(2) blockinterne Spaltenpermutation
(3) blockinterne Zeilenpermutation
(4) horizontale Blockpermutation
(5) vertikale Blockpermutation
(6) Transposition
(7) Spiegelung
(8) Rechtsdrehung

entstanden ist.

Wie würden Sie diese Aufgabe umsetzen?

Skizzieren Sie die Methoden, die Sie zusätzlich schreiben würden (Name und Beschreibung, wie in Kapitel 3, kein detaillierter Code).

Geben Sie, zum Beispiel in Pseudocode an, wie Sie Ihre Methoden miteinander verknüpfen würden.

Wir werden Ihren Vorschlag nach folgenden Kriterien beurteilen:

- Terminiert Ihre Methode?
- Arbeitet die Methode korrekt?
- Arbeitet Ihre Methode (zumindest in der Theorie) effizient, also vermeidet sie beispielsweise unnötige Schritte?

Sie dürfen Ihren Vorschlag als Teamvorschlag abgeben. Das heißt, Sie dürfen innerhalb Ihrer Gruppe zusammenarbeiten und eine gemeinsame Lösung abgeben.

# 6 Zusätzliche Infos

- 2018 05 08: Es kam die Frage auf, ob die Methoden für eine variable oder eine feste Gittergröße programmiert werden sollten. Eigentlich ist eine variable Größe besser, doch da der Meilenstein schon seit einer Woche veröffentlicht ist und ich Ihnen eine **grundlegende** Änderung der Aufgaben nicht zumuten möchte, verbleiben wir bei $9 \times 9$ - Sudokus, auf welche die Methoden arbeiten können müssen. Andere Gittergrößen müssen Ihre Methoden nicht bearbeiten können.

- 2018 05 08: Es kam die Frage auf, warum bei einer Blockpermutation der Bildvektor fünf Zellen besitzt und nicht drei Zellen, diese drei Zellen könnten jeweils aus einem der Diagonalblöcke sein.
  Dafür gibt es zwei Gründe: Erstens: Ich finde es leichter / zugänglicher, die fünf Zellen zu checken und sich daraus folgend zu überlegen, ob es zu einer vertikalen oder einer horizontalen Blockpermutation gekommen ist. Zweitens: Ich habe an diese Variante nicht gedacht. Diese Erkenntnis hat für die Aufgaben an sich keine Konsequenz, ich wollte Ihnen diese Information nur nicht vorenthalten.

- 2018 05 08: Es kam die Frage auf, ob die Singles sich nur auf eine Einheit beziehen oder auf das gesamte Gitter; vielleicht gibt es da Unklarheiten, daher nochmal zur Verdeutlichung: hierher

# 7 Änderungen zu vorherigen Versionen

- 2018 05 03 1522: In der vorherigen Version fehlte in den Interfaces 'Iso' jeweils die Methode, die jetzt an letzter Stelle fehlt. Danke für den Hinweis!
- 2018 05 03 2242: In der vorherigen Version hatten alle Methoden den Modifizierer `static`, was zu Fehlermeldungen führte. Diese sind jetzt entfernt. Wir bitten, diese Unachtsamkeit zu entschuldigen.
- 2018 05 06 0845: In der vorherigen Version führten kleinere und größere Unachtsamkeiten zu kleinerer und größerer Verwirrung. Marvin Pogoda schickte mir eine Zusammenfassung aus den Hinweisen, die sich in der zentralen E-Mailadresse sammelten (vielen Dank an Marvin und an die schreibenden Studierenden!)
  - ▷ `getMinimalNakedSingle` und `getNakedSingle` sind die gleichen Methoden, heißen aber unterschiedlich in der Erklärung und in den Interfaces. Wurde behoben.
  - ▷ Ähnliches Problem gab es bei `getHiddenPairCells` und anderen Methoden. Wurde behoben.
  - ▷ Der Satz „Eine **vertikale Blockpermutation** definiert sich analog zu einer blockweisen Spaltenpermutation“ wurde geändert zu „Eine **vertikale Blockpermutation** definiert sich analog zu einer horizontalen Blockpermutation.“ Analoges gilt für den Satz darunter.
  - ▷ Bei der Methode `void applyBlockPermutation(Grid grid, int[] image)` wurde `int[]` zu `Cell[]` geändert.
  - ▷ Der Satz „Die **n.-te Stelle** in einem Array ist der erste Speicherplatz, der in einem Array belegt werden kann.“ wurde korrigiert.
  - ▷ Abbildung 6: In Zelle $z_{1,6}$ im rechten Gitter wurde die 1 auf die 2 korrigiert.
  - ▷ Abbildung 7 hat auf Abbildung 15 verwiesen. Das war zum Lesen und Vergleichen recht umständlich. Das Sudoku aus Abbildung 15 wurde jetzt in Abbildung 7 kopiert.
  - ▷ Tippfehler „Bildverktor“ korrigiert.
- 2018 05 08 17 05: Es sind im Laufe der letzten Tage noch Fragen aufgekommen, woraufhin das ein oder andere sich ändert. Hier eine Liste, danke nochmal an Marvin für die meisten Hinweise, und Danke an die Studierenden mit den entsprechenden Fragen, die alle weiterbringen:
  - ▷ Kapitel 6 ist neu hinzugekommen.
  - ▷ `Cell[] getWhiteSpaces(Grid grid)` und verwandte Methoden wurden geändert zu
    `List<Cell> getWhiteSpaces(Grid grid)` (und analog die anderen), so, wie bereits in den Erläuterungen beschrieben, in den Interfaces aber noch nicht korrekt umgesetzt.
  - ▷ Tippfehler „Bildverktor“ an einer weiteren Stelle korrigiert.
  - ▷ `void applyBlockPermutation(Grid grid, int[] image)` war im Interface noch nicht korrigiert.
    Korrigiert: `void applyBlockPermutation(Grid grid, Cell[] image)`
  - ▷ Dieser Satz: „Dabei brauchen Sie nur den Fall zu betrachten, in dem die Einheit eine NakedSingle-Zelle enthält. Andere Fälle müssen Sie mit dieser Methode nicht abfangen.“ wurde geändert auf „Dabei brauchen Sie nur den Fall zu betrachten, in dem

die Einheit **mindestens** eine NakedSingle-Zelle enthält. Andere Fälle müssen Sie mit dieser Methode nicht abfangen.“ Ich vermute, dass es aus dem Kontext klar war (sonst wäre „minimal“ unnötig gewesen), dennoch ist das oberste Ziel, Verwirrung zu vermeiden.

▷ `Cell get<Unit>MinimalNakedSingleCell(Grid grid, Cell anchor):`

∗ Gibt die minimale NakedSingle-Zelle der Einheit zurück. Die Einheit wird durch `anchor` eindeutig festgelegt. Dabei brauchen Sie nur den Fall zu betrachten, in dem die Einheit mindestens eine NakedSingle-Zelle enthält. Andere Fälle müssen Sie mit dieser Methode nicht abfangen.

Was ist mit NakedSingle genau gemeint? Muss ich nur meine Einheit berücksichtigen und checken, ob durch die Informationen, die meine Einheit hergibt, eine NakedSingle-Zelle gefunden wird, oder muss ich meine Einheit und die beiden anderen Einheiten überprüfen, um festzustellen, ob eine NakedSingle-Zelle vorliegt?
Sie müssen letzteres machen. Beispiel: Möchten Sie in einem Block herausfinden, ob eine bestimmte Zelle eine NakedSingle-Zelle ist, so müssen Sie für diese Zelle den Block, die entsprechende Zeile und die entsprechende Spalte überprüfen, um diese Eigenschaft festzustellen. Analoges gilt für die anderen Strategien.

- 2018 05 09: In den Fragestunden kamen Fragen auf. An dieser Stelle Danke an Tim Steinbach.

  ▷ In 3.1.1 war bei (4) `boolean hasFullHouseUnit` falsch eingerückt bzw. doppelt.

  ▷ **Wichtig, da inhaltlich von Relevanz:** Neuer Kommentar am Ende von Abschnitt 3.4.

  ▷ Der Punkt

  2. Die Zahlen $\{1, \ldots, 9\}$ werden in die Zahlen $\{-1, 1, \ldots, 9\}$ abgebildet. Dabei müssen alle Zahlen aus $\{1, \ldots, 9\}$, die in dem Sudoku auftreten, auf eine Zahl aus $\{1, \ldots, 9\}$ abgebildet werden.

  wurde korrigiert, indem die `-1` entfernt wurde. Siehe Seite 8.

- 2018 05 10: Es kamen weitere Frage auf. An dieser Stelle Dank an Marvin Pogoda.

  ▷ Kleinigkeit: Einige Interfaces endeten **in den Überschriften** mit s. Korrigiert.

  ▷ Hinweis: die korrekte, englische Bezeichnung für eine Methode, die transponiert, wäre `transpose`, nicht `transponse`. Das ist mir im Eifer des Gefechts durchgegangen. Wir **bleiben trotz schmerzhaften Englisch bei der alten Bezeichnung**, um so wenig wie möglich Fehler zu erzeugen dadurch, dass sich die eigentliche Aufgabenstellung ändert. Mea Culpa.

  ▷ War **aus dem Kontext ersichtlich**, wird aber, um Verwirrung zu vermeiden, korrigiert: „Dieser **Bildvektor** $(a, b, c)$ ist so zu verstehen, dass in dem zu permutierenden Block die erste Zeile auf die a-te Stelle des Blocks, die zweite Zeile auf die b-te Stelle des Blocks und die dritte Zeile auf die c-te Zeile des Blocks abgebildet wird.“ wurde korrigiert zu „Dieser **Bildvektor** $(a, b, c)$ ist so zu verstehen, dass in dem zu permutierenden Block die erste Zeile auf die a-te Zeile (vorher stand hier Stelle) des Blocks, die zweite Zeile auf die b-te Zeile (vorher: Stelle) des Blocks und die dritte Zeile auf die c-te Zeile des Blocks abgebildet wird.“ Analog wurde die Spaltenversion korrigiert.

  ▷ **List**-Probleme: Ich habe den alten Text hier entfernt, weil er zu Verwirrung führen könnte. Bitte unten weiterlesen.

▷ `Grid.java` In dieser Datei gibt es noch TODOs, das heißt, Sie haben nicht die allerneuste Version zur Verfügung gestellt bekommen. Ich werde in den nächsten Tagen die aktuelle Version online stellen.

- Hinweise vom 20180511:

  ▷ **Wichtig!!!** - Nochmal zu `List`: Bevor man mich hier falsch versteht: Die Interfaces werden bitte nicht geändert! Das wollte ich damit nicht suggerieren. Bitte lesen Sie sich zu den Stichworten Polymorphie und abstrakte Klassen entsprechende Informationen durch und verwenden Sie 'intern' eine Klasse, die Ihnen passt. Zum Beispiel LinkedList, wie vorgeschlagen.

- 2018 05 14: Aufgekommene Fragen. Danke an Felix Breuer.

  ▷ (nicht mehr aktuell) Der Fehlerteufel hat sich eingeschlichen bei den Eigenschaften auf Seite 21. Ist nun hoffentlich korrigiert. Es fehlte die Erläuterung zu Eigenschaft 6, wobei dann auffiel, dass eigentlich Eigenschaft 2 nicht erklärt wurde. Alle Eigenschaften sind jetzt 'eins hochgerückt', und Eigenschaft 2 (Fullhouse) ist 'neu'.

  ▷ `getBlockMinimalNakedSingleCell(Grid grid, Cell anchor)` hieß zuvor `getBlockNakedSingleCell` und wich somit mit der Bezeichnung in der Erklärung ab. Wurde korrigiert.

- Hinweis vom 2018 05 15: Aufgekommene Frage. Danke an Marvin Pogoda.

  ▷ Marvin machte mich aufgrund einer Frage in den Fragestunden darauf aufmerksam, dass, je nachdem, wie man die Methode `<Unit>BasedSolvers` umsetzt, der Update-Teil obsolet sein könnte. Da ich mir bis dahin nur eine Lösung mit einer allgemeinen Kandidaten-Verwaltungs-Liste vorgestellt hatte, überraschte mich diese Frage zunächst. Daher nun diese Information: Wenn Ihre Methode genau die gleiche Art hat, wie unsere Methode vorzugehen und daraufhin die einzelnen Schritte / Grids identisch sind zu den von der oben vorgestellten Methode, **können** Sie den Update-Teil auch weglassen. Anders ausgedrückt: Wenn Ihre Rückgabe (also die Grid-Liste) das liefert, was auch unsere Rückgabe liefern würde, dann müssen Sie Update nicht programmieren, wenn das in Ihrer Umsetzung keinen Sinn macht.

- Hinweis vom 2018 05 18: Abschnitt 3.4 neu aufgesetzt. Mehr Informationen und Hinweise. Danke an Marvin.

- Hinweis vom 2018 05 28: Danke an Philipp Klinke.

  ▷ In dem Beispiel bei Update war etwas falsch beschrieben, sollte aus dem Kontext aber ersichtlich geworden sein. Dort stand: „Haben Sie beispielsweise in Schritt 5 das NakedPair 1 und 2 in Zeile 7 gefunden, sagen wir in Zelle $z_{7,1}$ und $z_{7,9}$, leeren Sie die Kandidatenlisten $L_{7,1}$ und $L_{7,9}$."
  Das wurde geändert zu: „Haben Sie beispielsweise in Schritt 5 das NakedPair 1 und 2 in Zeile 7 gefunden, sagen wir in Zelle $z_{7,1}$ und $z_{7,9}$, passen Sie die Kandidatenlisten $L_{7,1}$ und $L_{7,9}$ an."Nun sollte die Verwirrung aufgehoben sein.