

MEILENSTEIN 2

Inhaltsverzeichnis

1	Organisatorisches	1
1.1	Abgabe der Theorieaufgaben	1
1.2	Mitteilung der Individualaufgaben	2
1.3	Abgabe der Programmieraufgaben	2
1.4	Sonstiges	3
2	Individualaufgabe Typ „Col“	5
2.1	Interface ColSortable	5
2.2	Beschreibung der Methoden für „Col“	6
3	Individualaufgabe Typ „Row“	9
3.1	Interface RowSortable	9
3.2	Beschreibung der Methoden für „Row“	10
4	Individualaufgabe Typ „Block“	13
4.1	Interface BlockSortable	13
4.2	Beschreibung der Methoden für „Block“	14
5	Gruppenaufgabe	17
5.1	Interface SudokuGivens	18
5.2	Beschreibung der Methoden der Gruppenaufgabe	18
6	Beispiele	19
6.1	Übergabeparameter	19
6.2	Anordnungen	20
6.3	Nummern setzen	21
6.4	Eindeutigkeit, Minimalität und Redundanz	23

1 Organisatorisches

1.1 Abgabe der Theorieaufgaben

Die Theorieaufgaben, außer Javadoc, müssen handschriftlich bzw. ausgedruckt bis zum

09.Juli 2018 um 12 Uhr

im Briefkasten im Weyertal 80 abgegeben werden. Heften Sie bitte Ihre **Abgaben teamweise zusammen** oder stecken Sie sie in einem gemeinsamen Umschlag. Das **Javadoc** müssen Sie

definitiv mit hochladen. Ob Sie das Javadoc auch in ausgedruckter Form noch abgeben müssen, wird noch im Laufe der Bearbeitungszeit bekannt gegeben.

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.

1.2 Mitteilung der Individualaufgaben

Teilen Sie uns mit, wer in Ihrem Team welche Individualaufgabe macht. Das machen Sie bis spätestens

27.Juni 2018

durch eine Mitteilung an unsere zentrale E-Mailadresse, javaprogramm@uni-koeln.de.

1.3 Abgabe der Programmieraufgaben

Die Programmieraufgaben (individuell) müssen über die bereits bekannte Hochlademaske bis zum


09.07.2018, 12 Uhr,

eingereicht werden. Ab wann die Einreichung möglich ist, werden wir auf der Homepage bekannt geben.

- **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.
- 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.
- **Projektformat Meilenstein 2:** Betrachten Sie Meilenstein 2 als Erweiterung von Meilenstein 1. Das heißt, Sie können in dem unten angegebenen Projektformat bleiben und eventuell bereits geschriebene Methoden einfach weiter/wiederverwenden. Beachten Sie, dass die **zip**.-Datei **mindestens** ebendiese Ordnerstruktur aufweisen muss. Das bedeutet, dass auch weitere Ordner etc. eingefügt werden können.

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.
 - 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 **JavaTM 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.
 - Sie dürfen Ihre Lösungen aus Meilenstein 1 benutzen, um die ein oder andere Aufgabe aus Meilenstein 2 zu lösen. Wir empfehlen das natürlich nur, wenn Sie Ihre Lösung aus Meilenstein 1 gründlich getestet haben.
 - 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**.
 - Die zentrale **Emailadresse** javaprograp@uni-koeln.de ist nur noch für **organisatorische** Aspekte zu benutzen. Fachliche und inhaltliche Fragen werden in den Fragestunden geklärt. Die Tutoren sind angewiesen, Sie auf diese Fragestunden zu verweisen.

1.4 Sonstiges

- Wenn Sie die **offenen Fragestunden** in Anspruch nehmen möchten, so bringen Sie, wenn möglich, Ihren eigenen Laptop mit.
- 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 in den Fragestunden nach. Wie bisher auch, werden die sich angesammelten Fragen, insofern für eine größere Anzahl an Teilnehmenden relevant, an mich weitergeleitet und die Antworten in einer neuen Version des Meilensteins beantwortet. Nach Abgabe Ihrer Programme darauf aufmerksam zu machen, bringt niemandem etwas und wird dann auch in der Korrektur nicht mehr berücksichtigt werden. Beachten Sie auch, dass Hinweise zu Fehlern kurz vor Abgabe in Ihrer Verantwortung liegen. Bearbeiten Sie die Aufgaben nicht zu spät. Knappe Anfragen werden gegebenenfalls nicht mehr berücksichtigt.
- 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.
- Besonderen dank gilt Marvin und Tim, die sich vor allem bei inhaltlichen Aspekten und Aufgaben sehr engagiert zeigen! Vielen Dank!
- Für einige gute Anregungen und Ideen danke ich meinem Programmierpraktikum-Tutoren-Team: Felix, Karl, Marvin, Tim und Philippe.

2 Individualaufgabe Typ „Col“

Beachten Sie bitte, dass Sie **nicht** verpflichtet sind, „Col“ zu bearbeiten, wenn Sie bei Meilenstein 1 auch „Col“ bearbeitet haben.

(a) **Programmieraufgabe Typ „Col“** **50 Punkte**

Schreiben Sie eine Klasse namens `PuzzleCol`, welche das Interface `ColSortable` implementiert.

(b) **Theorieaufgabe „Javadoc“** **20 Punkte**

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. Beispielsweise würde Annie Hümpertz

`@author Huempertz, Annie`

in ihren Code schreiben. 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 generiertes) 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 hier gegebenen Definitionen als bekannt voraussetzen. Sie müssen die Begriffe also nicht nochmals in den Kommentaren erläutern.

(c) **Theorieaufgabe Typ „Col“** **30 Punkte**

Gegeben seien neun Spalten mit neun Zahlen. In jeder Spalte kommt jede Zahl zwischen 1 und 9 genau einmal vor. Weisen Sie nach, dass, wenn es eine Anordnung von 9 gegebenen Spalten gibt, so dass ein gültiges Sudoku entsteht, es noch mindestens $6^4 - 1$ weitere Möglichkeiten der Anordnung gibt, so dass ein gültiges Sudoku entsteht.

2.1 Interface ColSortable

Das Interface `ColSortable` besteht aus den Methoden

1. `Grid getColConflictFree(Grid grid)`
2. `boolean hasColConflictFree(Grid grid)`
3. `Grid getColSortRowBlock(Grid grid)`
4. `boolean hasColSortRowBlock(Grid grid)`
5. `Grid getColSudoku(Grid grid)`
6. `boolean hasColSudoku(Grid grid)`
7. `int putNumberRowBlock(Grid grid, int number)`

2.2 Beschreibung der Methoden für „Col“

1. Grid getColConflictFree(Grid grid)

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Falls es mindestens eine h -konfliktfreie Anordnung der gegebenen Spalten gibt, so wird diese als gefülltes Grid wiedergegeben. Wenn es mehrere solcher Anordnungen gibt, geben Sie nur eine zurück; welche, ist Ihnen überlassen. Falls es eine solche Anordnung nicht gibt, so wird ein Nullergrid zurückgegeben.

Siehe auch Abbildung 1.

- ▷ Wir nennen ein gefülltes Gitter **h -konfliktfrei** (h wie horizontal), wenn es folgende Eigenschaft erfüllt: Für je zwei nebeneinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die horizontal benachbart sind, also (direkt) nebeneinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Spalte nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter h -konfliktfrei. *Siehe auch Abbildung 1.*

- ▷ Ein **Nullergrid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

2. boolean hasColConflictFree(Grid grid)

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Spalten h -konfliktfrei anzuordnen. Ist eine h -konfliktfreie Anordnung möglich, ist der Rückgabewert `true`. Ansonsten `false`.

- ▷ Wir nennen ein gefülltes Gitter **h -konfliktfrei** (h wie horizontal), wenn es folgende Eigenschaft erfüllt: Für je zwei nebeneinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die horizontal benachbart sind, also (direkt) nebeneinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Spalte nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter h -konfliktfrei. *Siehe auch Abbildung 1.*

3. Grid getColSortRowBlock(Grid grid)

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergrid oder ein gefülltes Grid zurück. Die Methode gibt ein Nullergrid zurück, falls es nicht möglich ist, die vorgegebenen Spalten so anzuordnen, dass ein Grid entsteht, so dass in jeder Zeile und in jedem

Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses gefüllte Grid muss die Eigenschaft haben, dass es die gleichen Spalten hat wie das Inputgrid (allerdings nicht zwangsweise in der gleichen Reihenfolge) und dass in jeder Zeile und in jedem Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Wenn es mehrere Anordnungsmöglichkeiten der Spalten gibt (Sie also die Wahl zwischen mehreren gefüllten Grids für die Rückgabe haben), ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

4. `boolean hasColSortRowBlock(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob man die gegebenen Spalten so anordnen kann, dass in jeder Zeile und in jedem Block alle Zahlen zwischen 1 und 9 genau einmal vorkommen. Falls ja, wird `true` zurückgegeben. Falls nein, `false`.

Siehe auch Abbildung 2.

5. `Grid getColSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergid oder ein gefülltes Sudoku zurück. Die Methode gibt ein Nullergid zurück, falls es nicht möglich ist, die vorgegebenen Spalten so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses muss durch eine Anordnung der vorgegebenen Spalten entstanden sein und ein gefülltes Sudoku darstellen. Wenn es mehrere Möglichkeiten für ein zulässiges Sudoku nach obigen Vorgaben gibt, ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

6. `boolean hasColSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Spalten so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Falls es möglich ist, so ist der Rückgabewert `true`, ansonsten `false`.

Siehe auch Abbildung 2.

7. `int putNumberRowBlock(Grid grid, int number)`

- **Übergabeparameter:** An die Methode wird ein teilbefülltes Grid übergeben. Die Teilbefüllung hält sich an die unten genannte Regel.
 - ▷ Wir nennen ein Grid **teilbefüllt**, wenn es eine der folgenden Bedingungen erfüllt:
 - * Alle Zellen sind mit -1 belegt (es handelt sich also um ein „leeres“ Gitter).
 - * Einige Zellen sind bereits mit einer Zahl aus 1 bis 9 belegt, die restlichen Zellen sind mit -1 belegt.

Siehe auch Abbildung 3.
- **Aufgabe:** Die Methode soll die Anzahl der unterschiedlichen **number-Erweiterungen** zählen und zurückgeben.
 - ▷ Gegeben sei ein teilbefülltes Grid. Eine **number-Erweiterung** ist das Befüllen des Grids mit der Zahl **number**, so dass am Ende das Grid genau neun mal die Zahl **number** enthält. Dabei müssen die folgenden Regeln beachtet werden:
 - * Sie dürfen nur in die Zellen die Zahl **number** einsetzen, die eine -1 enthalten.
 - * In jeder Zeile und in jedem Block darf jeweils nur einmal die Zahl **number** enthalten sein.

Siehe auch Abbildungen 4, 5, 7 und 6.
 - ▷ Zwei **number-Erweiterungen** sind **unterschiedlich**, wenn es mindestens eine Zelle in einer **number-Erweiterung** gibt, die mit der Zahl **number** belegt ist und die in der anderen **number-Erweiterung** nicht mit der Zahl **number** belegt ist.

Siehe auch Abbildungen 4, 5, 7 und 6.

3 Individualaufgabe Typ „Row“

Beachten Sie bitte, dass Sie **nicht** verpflichtet sind, „Row“ zu bearbeiten, wenn Sie bei Meilenstein 1 auch „Row“ bearbeitet haben.

(a) **Programmieraufgabe Typ „Row“** **50 Punkte**

Schreiben Sie eine Klasse namens `PuzzleRow`, welche das Interface `RowSortable` implementiert.

(b) **Theorieaufgabe „Javadoc“** **20 Punkte**

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. Beispielsweise würde Annie Hümpertz

`@author Huempertz, Annie`

in ihren Code schreiben. 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 generiertes) 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 hier gegebenen Definitionen als bekannt voraussetzen. Sie müssen die Begriffe also nicht nochmals in den Kommentaren erläutern.

(c) **Theorieaufgabe Typ „Row“** **30 Punkte**

Gegeben seien neun Zeilen mit neun Zahlen. In jeder Zeile kommt jede Zahl zwischen 1 und 9 genau einmal vor. Weisen Sie nach, dass, wenn es eine Anordnung von 9 gegebenen Zeilen gibt, so dass ein gültiges Sudoku entsteht, es noch mindestens $6^4 - 1$ weitere Möglichkeiten der Anordnung gibt, so dass ein gültiges Sudoku entsteht.

3.1 Interface RowSortable

Das Interface `RowSortable` besteht aus den Methoden

1. `Grid getRowConflictFree(Grid grid)`
2. `boolean hasRowConflictFree(Grid grid)`
3. `Grid getRowSortColBlock(Grid grid)`
4. `boolean hasRowSortColBlock(Grid grid)`
5. `Grid getRowSudoku(Grid grid)`
6. `boolean hasRowSudoku(Grid grid)`
7. `int putNumberColBlock(Grid grid, int number)`

3.2 Beschreibung der Methoden für „Row“

1. Grid `getRowConflictFree(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Falls es mindestens eine *v*-konfliktfreie Anordnung der gegebenen Zeilen gibt, so wird diese als gefülltes Grid wiedergegeben. Wenn es mehrere solcher Anordnungen gibt, geben Sie nur eine zurück; welche, ist Ihnen überlassen. Falls es eine solche Anordnung nicht gibt, so wird ein Nullergrid zurückgegeben.

Siehe auch Abbildung 1.

- ▷ Wir nennen ein gefülltes Gitter ***v*-konfliktfrei** (*v* wie vertikal), wenn es folgende Eigenschaft erfüllt: Für je zwei übereinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die vertikal benachbart sind, also (direkt) übereinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Zeile nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter *v*-konfliktfrei. *Siehe auch Abbildung 1.*

- ▷ Ein **Nullergrid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

2. boolean `hasRowConflictFree(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Zeilen *v*-konfliktfrei anzuordnen. Ist eine *v*-konfliktfreie Anordnung möglich, ist der Rückgabewert `true`. Ansonsten `false`.

- ▷ Wir nennen ein gefülltes Gitter ***v*-konfliktfrei** (*v* wie vertikal), wenn es folgende Eigenschaft erfüllt: Für je zwei übereinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die vertikal benachbart sind, also (direkt) übereinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Zeile nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter *v*-konfliktfrei. *Siehe auch Abbildung 1.*

3. Grid `getRowSortColBlock(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergrid oder ein gefülltes Grid zurück. Die Methode gibt ein Nullergrid zurück, falls es nicht möglich ist, die vorgegebenen Zeilen so anzuordnen, dass ein Grid entsteht, so dass in jeder Spalte und in jedem

Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses gefüllte Grid muss die Eigenschaft haben, dass es die gleichen Zeilen hat wie das Inputgrid (allerdings nicht zwangsweise in der gleichen Reihenfolge) und dass in jeder Spalte und in jedem Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Wenn es mehrere Anordnungsmöglichkeiten der Zeilen gibt (Sie also die Wahl zwischen mehreren gefüllten Grids für die Rückgabe haben), ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

4. `boolean hasRowSortColBlock(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob man die gegebenen Zeilen so anordnen kann, dass in jeder Spalte und in jedem Block alle Zahlen zwischen 1 und 9 genau einmal vorkommen. Falls ja, wird `true` zurückgegeben. Falls nein, `false`.

Siehe auch Abbildung 2.

5. `Grid getRowSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergid oder ein gefülltes Sudoku zurück. Die Methode gibt ein Nullergid zurück, falls es nicht möglich ist, die vorgegebenen Zeilen so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses muss durch eine Anordnung der vorgegebenen Zeilen entstanden sein und ein gefülltes Sudoku darstellen. Wenn es mehrere Möglichkeiten für ein zulässiges Sudoku nach obigen Vorgaben gibt, ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

6. `boolean hasRowSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Zeilen ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine einzelne Zeile weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Zeilen so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Falls es möglich ist, so ist der Rückgabewert `true`, ansonsten `false`.

Siehe auch Abbildung 2.

7. `int putNumberColBlock(Grid grid, int number)`

- **Übergabeparameter:** An die Methode wird ein teilbefülltes Grid übergeben. Die Teilbefüllung hält sich an die unten genannte Regel.
 - ▷ Wir nennen ein Grid **teilbefüllt**, wenn es eine der folgenden Bedingungen erfüllt:
 - * Alle Zellen sind mit -1 belegt (es handelt sich also um ein „leeres“ Gitter).
 - * Einige Zellen sind bereits mit einer Zahl aus 1 bis 9 belegt, die restlichen Zellen sind mit -1 belegt.

Siehe auch Abbildung 3.
- **Aufgabe:** Die Methode soll die Anzahl der unterschiedlichen **number-Erweiterungen** zählen und zurückgeben.
 - ▷ Gegeben sei ein teilbefülltes Grid. Eine **number-Erweiterung** ist das Befüllen des Grids mit der Zahl **number**, so dass am Ende das Grid genau neun mal die Zahl **number** enthält. Dabei müssen die folgenden Regeln beachtet werden:
 - * Sie dürfen nur in die Zellen die Zahl **number** einsetzen, die eine -1 enthalten.
 - * In jeder Spalte und in jedem Block darf jeweils nur einmal die Zahl **number** enthalten sein.

Siehe auch Abbildungen 4, 5, 7 und 6.
 - ▷ Zwei **number-Erweiterungen** sind **unterschiedlich**, wenn es mindestens eine Zelle in einer **number-Erweiterung** gibt, die mit der Zahl **number** belegt ist und die in der anderen **number-Erweiterung** nicht mit der Zahl **number** belegt ist.

Siehe auch Abbildungen 4, 5, 7 und 6.

4 Individualaufgabe Typ „Block“

Beachten Sie bitte, dass Sie **nicht** verpflichtet sind, „Block“ zu bearbeiten, wenn Sie bei Meilenstein 1 auch „Block“ bearbeitet haben.

(a) **Programmieraufgabe Typ „Block“** **50 Punkte**

Schreiben Sie eine Klasse namens `PuzzleBlock`, welche das Interface `BlockSortable` implementiert.

(b) **Theorieaufgabe „Javadoc“** **20 Punkte**

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. Beispielsweise würde Annie Hümpertz

`@author Huempertz, Annie`

in ihren Code schreiben. 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 generiertes) 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 hier gegebenen Definitionen als bekannt voraussetzen. Sie müssen die Begriffe also nicht nochmals in den Kommentaren erläutern.

(c) **Theorieaufgabe Typ „Block“** **30 Punkte**

Gegeben seien neun Blöcke mit neun Zahlen. In jedem Block kommt jede Zahl zwischen 1 und 9 genau einmal vor. Weisen Sie nach, dass, wenn es eine Anordnung von 9 gegebenen Blöcke gibt, so dass ein gültiges Sudoku entsteht, es noch mindestens 35 weitere Möglichkeiten der Anordnung gibt, so dass ein gültiges Sudoku entsteht.

4.1 Interface `BlockSortable`

Das Interface `BlockSortable` besteht aus den Methoden

1. `Grid getBlockConflictFree(Grid grid)`
2. `boolean hasBlockConflictFree(Grid grid)`
3. `Grid getBlockSortColRow(Grid grid)`
4. `boolean hasBlockSortColRow(Grid grid)`
5. `Grid getBlockSudoku(Grid grid)`
6. `boolean hasBlockSudoku(Grid grid)`
7. `int putNumberColRow(Grid grid, int number)`

4.2 Beschreibung der Methoden für „Block“

1. Grid `getBlockConflictFree(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jeder der neun Blöcke ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für einen einzelnen Block weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

Sie dürfen allerdings in dieser Methode davon ausgehen, dass die Blöcke *h*-konfliktfrei belegt sind.

- **Aufgabe:** Falls es mindestens eine *h*-konfliktfreie Anordnung der gegebenen Blöcke gibt, so wird diese als gefülltes Grid wiedergegeben. Beachten Sie dazu bitte den Hinweis bei `hasBlockConflictFree`. Wenn es mehrere solcher Anordnungen gibt, geben Sie nur eine zurück; welche, ist Ihnen überlassen. Falls es keine solche Anordnung nicht gibt, so wird ein Nullergrid zurückgegeben.

▷ Wir nennen ein gefülltes Gitter ***h*-konfliktfrei** (*h* wie horizontal), wenn es folgende Eigenschaft erfüllt: Für je zwei nebeneinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die horizontal benachbart sind, also (direkt) nebeneinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Spalte nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter *h*-konfliktfrei. *Siehe auch Abbildung 1.*

▷ Ein **Nullergrid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

2. boolean `hasBlockConflictFree(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jeder der neun Blöcke ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für einen einzelnen Block weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

Sie dürfen allerdings in dieser Methode davon ausgehen, dass die Blöcke *h*-konfliktfrei belegt sind.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Blöcke *h*-konfliktfrei anzuordnen.

▷ Wir nennen ein gefülltes Gitter ***h*-konfliktfrei** (*h* wie horizontal), wenn es folgende Eigenschaft erfüllt: Für je zwei nebeneinander stehende Zellen muss gelten, dass sie nicht die gleichen Zahlen beinhalten. Es darf also in dem gesamten Gitter keine zwei gleichen Zahlen geben, die horizontal benachbart sind, also (direkt) nebeneinander in dem Gitter stehen.

Hinweis: Stehen in einem Gitter in der ersten Spalte nur 1en, in der zweiten nur 2en, und so weiter, so ist dieses Gitter *h*-konfliktfrei. *Siehe auch Abbildung 1.*

- **Hinweis:** In dieser Methode müssen Sie also die neun gegebenen Blöcke so auf ein Grid verteilen, dass der seitliche Rand (links oder rechts) eines Blocks nicht so mit dem seitlichen Rand eines anderen Blocks in Berührung kommt, dass zwei gleiche Zahlen in direkt miteinander benachbarten Zellen stehen.

3. Grid `getBlockSortColRow(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jede der neun Spalten ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für eine

einzelne Spalte weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergrid oder ein gefülltes Grid zurück. Die Methode gibt ein Nullergrid zurück, falls es nicht möglich ist, die vorgegebenen Blöcke so anzuordnen, dass ein Grid entsteht, so dass in jeder Zeile und in jeder Spalte jede Zahl zwischen 1 und 9 genau einmal vorkommt. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses gefüllte Grid muss die Eigenschaft haben, dass es die gleichen Blöcke hat wie das Inputgrid (allerdings nicht zwangsweise in der gleichen Reihenfolge) und dass in jeder Zeile und in jeder Spalte jede Zahl zwischen 1 und 9 genau einmal vorkommt. Wenn es mehrere Anordnungsmöglichkeiten der Blöcke gibt (Sie also die Wahl zwischen mehreren gefüllten Grids für die Rückgabe haben), ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergrid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

4. `boolean hasBlockSortColRow(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jeder der neun Blöcke ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für einen einzelnen Block weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob man die gegebenen Blöcke so anordnen kann, dass in jeder Spalte und in jeder Zeile alle Zahlen zwischen 1 und 9 genau einmal vorkommen. Falls ja, wird `true` zurückgegeben. Falls nein, `false`.

Siehe auch Abbildung 2.

5. `Grid getBlockSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jeder der neun Blöcke ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für einen einzelnen Block weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode gibt entweder ein Nullergrid oder ein gefülltes Sudoku zurück. Die Methode gibt ein Nullergrid zurück, falls es nicht möglich ist, die vorgegebenen Blöcke so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Anderenfalls gibt die Methode ein gefülltes Grid zurück. Dieses muss durch eine Anordnung der vorgegebenen Blöcke entstanden sein und ein gefülltes Sudoku darstellen. Wenn es mehrere Möglichkeiten für ein zulässiges Sudoku nach obigen Vorgaben gibt, ist es Ihnen überlassen, welche Belegung Sie zurückgeben.

Siehe auch Abbildung 2.

▷ Ein **Nullergrid** ist ein Grid, in dem in jeder Zelle eine 0 steht.

6. `boolean hasBlockSudoku(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe das Grid `grid`. Jeder der neun Blöcke ist komplett mit Zahlen zwischen 1 und 9 belegt. Dabei muss für einen einzelnen Block weder gelten, dass jede Zahl nur einmal vorkommt, noch muss gelten, dass jede Zahl vorkommt.

Beispiele für gültige Inputs finden Sie in Abbildung 1.

- **Aufgabe:** Die Methode entscheidet, ob es möglich ist, die neun gegebenen Blöcke so anzuordnen, dass ein zulässig gefülltes Sudoku entsteht. Falls es möglich ist, so ist der Rückgabewert `true`, ansonsten `false`.

Siehe auch Abbildung 2.

7. `int putNumberColRow(Grid grid, int number)`

- **Übergabeparameter:** An die Methode wird ein teilbefülltes Grid übergeben. Die Teilbefüllung hält sich an die unten genannte Regel.

▷ Wir nennen ein Grid **teilbefüllt**, wenn es eine der folgenden Bedingungen erfüllt:

- * Alle Zellen sind mit `-1` belegt (es handelt sich also um ein „leeres“ Gitter).
- * Einige Zellen sind bereits mit einer Zahl aus 1 bis 9 belegt, die restlichen Zellen sind mit `-1` belegt.

Siehe auch Abbildung 3.

- **Aufgabe:** Die Methode soll die Anzahl der unterschiedlichen **number**-Erweiterungen zählen und zurückgeben.

▷ Gegeben sei ein teilbefülltes Grid. Eine **number-Erweiterung** ist das Befüllen des Grids mit der Zahl **number**, so dass am Ende das Grid genau neun mal die Zahl **number** enthält. Dabei müssen die folgenden Regeln beachtet werden:

- * Sie dürfen nur in die Zellen die Zahl **number** einsetzen, die eine `-1` enthalten.
- * In jeder Spalte und in jeder Zeile darf jeweils nur einmal die Zahl **number** enthalten sein.

Siehe auch Abbildungen 4, 5, 7 und 6.

▷ Zwei **number-Erweiterungen** sind **unterschiedlich**, wenn es mindestens eine Zelle in einer **number-Erweiterung** gibt, die mit der Zahl **number** belegt ist und die in der anderen **number-Erweiterung** nicht mit der Zahl **number** belegt ist.

Siehe auch Abbildungen 4, 5, 7 und 6.

5 Gruppenaufgabe

- (a) **Programmieraufgabe „Gruppe“** **50 Punkte** Schreiben Sie eine Klasse namens `MinSudoku`, welche das Interface `SudokuGivens` implementiert.

- (b) **Theorieaufgabe „Javadoc“** **20 Punkte**
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. Beispielsweise würde Annie Hümpertz

`@author Huempertz, Annie`

in ihren Code schreiben. 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 generiertes) 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 hier gegebenen Definitionen als bekannt voraussetzen. Sie müssen die Begriffe also nicht nochmals in den Kommentaren erläutern.

- (c) **Theorieaufgabe** **30 Punkte**
`Sudoku-Gitter` kann man auch graphentheoretisch betrachten. Für eine Auffrischung empfiehlt sich ein Blick in die Informatik I und II sowie in die entsprechenden Lehrbücher.

Nehmen wir an, es sind neun Spalten wie in der Methode

`hasColConflictFree(Grid grid)`

gegeben. Nehmen wir weiterhin an, dass Ihnen eine Methode `hasPath` zur Verfügung steht, die entscheiden kann, ob ein Graph eine Knotenfolge von neun Knoten enthält, so dass der erste Knoten mit dem zweiten, der zweite mit dem dritten, der dritte mit dem vierten und so weiter verbunden ist. Dabei ist es unerheblich, ob die Knoten untereinander noch mehr Verbindungen haben. Falls eine solche Knotenfolge existiert, so gibt `hasPath` den Wert `true` zurück, sonst `false`.

Bauen Sie einen Graphen so auf, dass Folgendes gilt: Der Rückgabewert von `hasPath` stimmt mit dem Rückgabewert von `hasColConflictFree(Grid grid)` überein. In anderen Worten: Modellieren Sie das Problem aus `hasColConflictFree(Grid grid)` in Form eines Graphen, so dass der Graph genau dann eine wie oben beschriebene Knotenfolge hat, wenn es möglich ist, eine Anordnung der Spalten so hinzubekommen, so dass die Anordnung *h*-konfliktfrei ist.

Sie müssen keinen Pseudocode hinschreiben. Notieren Sie in Ihren eigenen Worten („Prosa“), wie Sie den Graphen modellieren würden, und warum dann die oben verlangte Bedingung gilt. Im Übrigen nennt man die oben beschriebene Knotenfolge einen (nicht induzierten) Pfad auf 9 Knoten.

Ist Ihre Konstruktion auf andere Einheiten übertragbar?

5.1 Interface SudokuGivens

Das Interface `SudokuGivens` enthält folgende, abstrakte Methoden:

1. `boolean isMinimalSudoku(Grid grid)`
2. `Cell getMinimalObsoleteGiven(Grid grid)`

5.2 Beschreibung der Methoden der Gruppenaufgabe

1. `boolean isMinimalSudoku(Grid grid);`

- **Übergabeparameter:** Die Methode erhält als Eingabe ein teilbefülltes Grid. Die mit `-1` gefüllten Zellen sind die „leeren“, also unbelegten Zellen.

Siehe auch Abbildung 3.

- **Aufgabe:** Die Methode entscheidet, ob das gegebene Sudoku minimal ist. Falls ja, wird `true` zurückgegeben, falls nein, `false`. Sie dürfen davon ausgehen, dass Sie nur zulässig teilbefüllte Sudokus als Eingabe bekommen.

▷ Wir nennen ein Sudoku **eindeutig lösbar**, wenn es nur eine einzige Lösung gibt. Insbesondere sind vollständig gefüllte Sudokus eindeutig lösbar.

- * Hinweis: Die Reihenfolge, in der man die noch nicht belegten Zellen ausfüllen muss, muss nicht eindeutig sein. Ist das Sudoku beispielsweise voll bis auf zwei Zellen, sagen wir $z_{1,2}$ und $z_{1,3}$, so können diese Zellen nur auf eine eindeutige Art und Weise belegt werden. Ob man dabei erst $z_{1,2}$ und dann $z_{1,3}$ befüllt, ist für die Eindeutigkeit unerheblich.

Siehe auch Abbildungen 8 und 9.

▷ Wir nennen in einem Sudoku eine eingetragene Zahl zwischen 1 und 9 ein **Given**.

- * Hinweis: Ein Sudoku, wie wir es zum Rätseln und Lösen aus Zeitungen etc. kennen, ist im Wesentlichen ein Gitter mit ein paar Givens, also ein paar gegebenen Zahlen.

Siehe auch Abbildungen 8 und 9.

▷ Wir nennen ein Sudoku **minimal**, wenn es eindeutig lösbar ist und die Wegnahme eines beliebigen Givens daraus ein Sudoku macht, welches **mindestens** zwei Lösungen hat.

Siehe auch Abbildung 10.

2. `Cell getMinimalObsoleteGiven(Grid grid)`

- **Übergabeparameter:** Die Methode erhält als Eingabe ein teilbefülltes Grid. Die mit `-1` gefüllten Zellen sind die „leeren“, also unbefüllten Zellen.

Siehe auch Abbildung 3.

- **Aufgabe:** Die Methode stellt fest, ob sich in dem Grid mindestens ein obsoletes Given befindet. Falls nicht, gibt die Methode die Zelle $z_{1,1}$ mit dem Wert `-1` wieder. Falls es mindestens ein obsoletes Given in dem Grid gibt, dann gibt die Methode die Zelle mit minimalem Index zurück, die ein obsoletes Given enthält.

▷ Wir nennen ein Given **obsolet**, wenn das Entfernen des Givens die Lösungsmenge unverändert lässt.

- * Hinweis: Anders ausgedrückt: Wenn wir eine gegebene Zahl aus dem Gitter entfernen, und es kommt dadurch keine einzige neue Lösung hinzu, dann war diese gegebene Zahl (das Given) obsolet.

Siehe auch Abbildung 11.

6 Beispiele

6.1 Übergabeparameter

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	3	3	3	3	3	4	5	9
2	3	3	3	3	3	4	5	9
2	3	3	3	3	3	4	5	9
8	8	8	8	4	4	7	8	7
2	3	3	3	3	3	4	5	9

1	2	1	1	2	1	1	2	1
2	1	2	2	1	2	2	1	2
1	2	1	1	2	1	1	2	1
1	2	1	1	2	1	1	2	1
2	1	2	2	1	2	2	1	2
1	2	1	1	2	1	1	2	1
1	2	1	1	2	1	1	2	1
2	1	2	2	1	2	2	1	2
1	2	1	1	2	1	1	2	1

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

Abbildung 1: Beispiele für ein komplett befülltes, übergebenes Input-Grid. Es gibt keine „leeren“ Zellen, ansonsten ist die Befüllung mit den Zahlen 1 bis 9 beliebig.

Das linke Gitter und das mittlere Gitter sind zum Beispiel wegen der Zellen $z_{3,3}$ und $z_{3,4}$ nicht h -konfliktfrei belegt. Das rechte Gitter ist h -konfliktfrei belegt. Keins der Gitter ist v -konfliktfrei belegt. Hätte eines der Gitter ein Schachbrettmuster, so wäre dies sowohl h - als auch v -konfliktfrei belegt.

Bei der Methode `getColConflictFree` würde bei Input des linken Gitters und mittleren Gitters ein Nullgrid zurück gegeben werden. Bei Input des rechten Gitters würde die bereits vorgegebene Belegung eine gültige Belegung sein; man könnte also das Grid unverändert wieder in den Output geben. Man könnte aber genauso die erste und zweite Spalte vertauschen. Auch dies wäre eine gültige Antwort.

6.2 Anordnungen

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

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

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

Abbildung 2: Das linke Grid ist ein Inputgrid, bei dem es möglich ist, die Spalten so anzuordnen, dass das Gitter danach so befüllt ist, dass in jeder Zeile und in jedem Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Dargestellt ist hier nur eine von mehreren möglichen Anordnungen. Dennoch ist es bei diesem Grid nicht möglich, die Spalten so anzuordnen, dass ein gültiges Sudoku entsteht.

Das mittlere Grid ist ein Inputgrid, bei dem es möglich ist, die Zeilen so anzuordnen, dass das Gitter danach so befüllt ist, dass in jeder Spalte und in jedem Block jede Zahl zwischen 1 und 9 genau einmal vorkommt. Dargestellt ist hier nur eine von mehreren möglichen Anordnungen. Dennoch ist es bei diesem Grid nicht möglich, die Zeilen so anzuordnen, dass ein gültiges Sudoku entsteht.

Das rechte Grid ist ein Inputgrid, bei dem es möglich ist, die Blöcke so anzuordnen, dass das Gitter danach so befüllt ist, dass in jeder Zeile und in jeder Spalte jede Zahl zwischen 1 und 9 genau einmal vorkommt. Dargestellt ist hier nur eine von mehreren möglichen Anordnungen. Dennoch ist es bei diesem Grid nicht möglich, die Blöcke so anzuordnen, dass ein gültiges Sudoku entsteht.

6.3 Nummern setzen

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

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

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

Abbildung 3: Das linke Grid ist ein „leeres“ Grid und somit per Definition ein teilbefülltes Grid. Das mittlere Gitter ist ebenfalls ein teilbefülltes Grid. Das rechte Grid ist kein teilbefülltes Grid, denn es enthält keine einzige -1.

1								
	1							
		1						
			1					
				1				
					1			
						1		
							1	
								1

1								
	1							
		1						
			1					
				1				
					1			
						1		
							1	
								1

Abbildung 4: Das mittlere Grid ist eine 1-Erweiterung des linken Grids, die der Regel folgt, dass in jeder Spalte und in jeder Zeile die Zahl 1 nur einmal vorkommen darf. Das rechte Grid stellt eine andere 1-Erweiterung nach den gleichen Regeln dar. Um das Grid übersichtlicher zu gestalten, entsprechen alle leeren Zellen einer Zelle mit -1.

1							
	2					1	
		7	7	7			
				1			
				3			1
	1						
		1					
		4			8		

1							
	2					1	
		7	7	7	1		
				1			
				3			1
	1						
		1					
		4	1		8		
					1		

1							
	2					1	
		7	7	7	1		
				1			
				3			1
	1						
		1					
		4	1		8		
					1		

Abbildung 5: Das mittlere Grid ist eine 1-Erweiterung des linken Grids, die der Regel folgt, dass in jeder Spalte und in jeder Zeile die Zahl 1 nur einmal vorkommen darf. Das rechte Grid stellt eine andere 1-Erweiterung nach den gleichen Regeln dar. Um das Grid übersichtlicher zu gestalten, entsprechen alle leeren Zellen einer Zelle mit -1.

5							
	2					5	
		7	7	7			
				5			
				3			5
	5						
		5					
		4					

5							
	2					5	
		7	7	7	5		
				5			
				3			5
	5						
		5					
		4	5				
					5		

5							
	2					5	
		7	7	7	5		
				5			
				3			5
	5						
		5					
		4	5			5	

Abbildung 6: Das mittlere Grid ist eine 5-Erweiterung des linken Grids, die der Regel folgt, dass in jeder Spalte und in jedem Block die Zahl 5 nur einmal vorkommen darf. Das rechte Grid stellt eine andere 5-Erweiterung nach den gleichen Regeln dar. Um das Grid übersichtlicher zu gestalten, entsprechen alle leeren Zellen einer Zelle mit -1.

6							
	2					6	
		7	7	7			
				6			
				3			6
	6						
		6					
		4					

6							
	2					6	
		7	7	7	6		
				6			
				3			6
	6						
		6					
		4	6				
					6		

6							
	2					6	
		7	7	7	6		
				6			
				3			6
	6						
		6					
		4	6				
						6	

Abbildung 7: Das mittlere Grid ist eine 6-Erweiterung des linken Grids, die der Regel folgt, dass in jeder Zeile und in jedem Block die Zahl 6 nur einmal vorkommen darf. Das rechte Grid stellt eine andere 6-Erweiterung nach den gleichen Regeln dar. Um das Grid übersichtlicher zu gestalten, entsprechen alle leeren Zellen einer Zelle mit -1.

6.4 Eindeutigkeit, Minimalität und Redundanz

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

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

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

Abbildung 8: Das linke Grid stellt ein vollständig gelöstes Sudoku dar und ist somit per Definition bereits eindeutig lösbar. Das mittlere Grid stellt ebenfalls ein eindeutig lösbares Sudoku dar (wir haben aus Gründen der Übersichtlichkeit die -1 nicht hingeschrieben). Die einzig mögliche Lösung sehen Sie im linken Grid. Das rechts Sudoku ist ein eindeutig lösbares Sudoku mit nur 17 Givens, entnommen aus „Sudoku Squares and Polynomials“ von Herzberg und Murty.

	4	7	3	6	9		5	8
	5	8		4	7	3	6	9
3	6	9		5	8		4	7
7		4	9	3	6	8		5
8		5	7		4	9	3	6
9	3	6	8		5	7		4
4	7		6	9	3	5	8	
5	8		4	7		6	9	3
6	9	3	5	8		4	7	

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

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

Abbildung 9: Nicht eindeutig lösbares Sudoku (ganz links) mit zwei möglichen Lösungen (es gibt vermutlich noch mehr).

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

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

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

Abbildung 10: Das linke Sudoku ist ein minimales Sudoku. Die leeren Zellen sind nur auf eine Art zu füllen, wie man im mittleren Gitter sieht. Wäre $z_{1,1}$ **nicht** mit einer 1 belegt, so wären mindestens zwei Möglichkeiten gegeben, das Sudoku zu füllen; diese Möglichkeiten sind in dem mittleren und dem rechten Sudoku abzulesen. Auch die 1 in $z_{5,5}$ und $z_{1,1}$ sind obsolet.

1								
				1				
								1
	1							
			1					
							1	
		1						
					1			
						1		

1								
				1				
								1
	1							
			1					
							1	
		1						
					1			

				1				
								1
	1							
							1	
		1						
					1			

Abbildung 11: Die 1 in Zelle $z_{9,7}$ im linken Grid ist obsolet. Entfernt man diese 1, so resultiert daraus das mittlere Gitter. Da wir im mittleren Gitter ohnehin in jeder Lösung die letzte 1 immer in $z_{9,7}$ schreiben müssen, ändert sich die Lösungsmenge durch Entfernen der 1 nicht. Kein Given im rechten Grid ist obsolet.