IWI Fakultät für Informatik
und Wirtschaftsinformatik

Sebastian Hesse, M.Sc.
Prof. Dr. Heiko Körner

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES
Näher dran.

Pflichtprogrammieraufgabe zum Kurs Programmieren mit Java (MKI)

Das Spiel *Mastermind* (siehe `de.wikipedia.org/wiki/Mastermind_(Spiel)`) war beginnend ab 1970 eines der erfolgreichsten Logikspiele und erfreut heute noch viele Kinder und Jugendliche — und jetzt auch Sie, denn Sie sollen es in Java nachprogrammieren! Ursprünglich werden in *Mastermind* farbige kleine Stifte verwendet, wir werden jedoch Ziffern dafür benutzen.

Mastermind wird von zwei Personen gespielt. Die eine Person legt zu Beginn einen geheim gehaltenen vierstelligen Code fest, z.B. 3132. Die andere Person soll diesen erraten und schlägt dafür mehrmals eigene Codes vor, z.B. 1234. Dieser wird von der ersten Person auf seine einzelnen Ziffern hin untersucht und wie folgt bewertet:

- Steht eine Ziffer an der richtigen Position (dies ist hier bei der Ziffer 3 der Fall), so gibt es dafür einen schwarzen Stift.
- Sind darüber hinaus Ziffern zwar vorhanden, aber falsch positioniert (hier die 1 und die 2), so gibt es dafür einen weißen Stift. Auch die Ziffer 3 ist nochmals im Gehimcode an führender Position vertreten, da sie als schwarzer Treffer aber bereits „verbraucht“ wurde, zählt sie hier nicht nochmals mit.

Im Originalspiel werden die schwarzen und weißen Bewertungsstifte auf ein Spielbrett eingesteckt. Wir nehmen hier stattdessen auf dem Bildschirm eine Ausgabe der Form s–w vor, wobei s die Anzahl der schwarzen und w die Anzahl der weißen Stifte angibt. Hier wäre die Ausgabe also 1–2 gewesen. An der Bewertung 4–0 erkennt man, dass der Code richtig erraten wurde.

Alle Versuche des ratendes Spielers mit den zugehörigen Bewertungen werden protokolliert. Ein möglicher Spielverlauf könnte also wie folgt aussehen:

```
Willkommen zu Mastermind - und los geht's!
Gib bitte Deinen Versuch #1 ein: 1234
Bislang hast Du so gespielt:
Dein Versuch #1: 1234 Bewertung: 0-2
Gib bitte Deinen Versuch #2 ein: 5678
Bislang hast Du so gespielt:
Dein Versuch #2: 5678 Bewertung: 1-0
Dein Versuch #1: 1234 Bewertung: 0-2
Gib bitte Deinen Versuch #3 ein: 2379
Bislang hast Du so gespielt:
Dein Versuch #3: 2379 Bewertung: 0-0
Dein Versuch #2: 5678 Bewertung: 1-0
Dein Versuch #1: 1234 Bewertung: 0-2
                ...
```

So gibt der ratende Spieler Versuch für Versuch ein. Am Ende (bei dem folgenden Spielstand sind nur noch die Codes 4118 und 4188 möglich — warum?) hat er dann Glück:

```
Bislang hast Du so gespielt:
Dein Versuch #5: 6641 Bewertung: 0-2
Dein Versuch #4: 5145 Bewertung: 1-1
Dein Versuch #3: 2379 Bewertung: 0-0
Dein Versuch #2: 5678 Bewertung: 1-0
Dein Versuch #1: 1234 Bewertung: 0-2
Gib bitte Deinen Versuch #6 ein: 4188
Glückwunsch! - Du hast gewonnen!
```

Wir wollen *Mastermind* noch ein wenig allgemeiner erstellen und folgende Erweiterungen zulassen:

- Die Codelänge (zwischen 1 und 9) kann vorgegeben werden. Man kann also nicht nur mit 4–stelligen, sondern auch z.B. mit 7–stelligen Codes spielen.
- Die Anzahl der Farben (hier also der Bereich der erlaubten Ziffern) kann ebenfalls zwischen 1 und 9 vorgegeben werden. Bei beispielsweise sechs Farben sind dann nur die Ziffern von 1 bis 6 erlaubt (die Ziffer 0 ist grundsätzlich ungültig).

Beachten Sie, dass Sie mit diesen Rahmenbedingungen einen Versuch und/oder den Geheimcode in einer einfachen `int`–Zahl abspeichern können.

Aufgabe 1: Programmieren Sie eine Modell–Klasse für das Spiel *Mastermind*. Sie soll über folgende `public`–Methoden verfügen:

- Der Konstruktor nimmt als Parameter die Codelänge und die Farbanzahl entgegen. Unzulässige Angaben werden kommentarlos durch die Standardwerte des Originalspiels (vier Ziffern und sechs Farben) ersetzt. Im Konstruktor wird auch der Geheimcode generiert.
- Ein zweiter parameterloser Standard–Konstruktor verwendet ebenfalls vier Ziffern und sechs Farben.
- `boolean speichereNaechstenVersuch(int versuch)` nimmt einen Versuch entgegen und speichert ihn ab, so dass er jederzeit später wieder aufgerufen werden kann (siehe nachfolgende Methode `rufeVersuchAb`). Die Speicherung darf nur bei einem gültigen Versuch erfolgen (es treten also keine unerlaubten Ziffern auf und auch die Codelänge muss stimmen). Bei einem ungültigen Versuch ist `false` zurückzugeben, ansonsten `true`.
- Sie dürfen die Anzahl der verfügbaren Versuche auf z.B. 1000 limitieren, was für unsere Implementierung ausreichen sollte. Legen Sie dazu eine Konstante `private final int maxVersuche = 1000;` an und erstellen Sie den restlichen Code in Abhängigkeit von diesem Wert, so dass man die Versuchsanzahl später trotzdem noch erhöhen kann.
- `int rufeAnzahlVersucheAb()` gibt die Anzahl der bisherigen Versuche zurück.
- `int rufeVersuchAb(int index)` gibt den `index`–ten Versuch zurück. Diese sind ab 1 durchnummeriert, d.h. bei bislang z.B. sieben Versuchen gibt es die Versuche mit den Nummern 1 bis 7. Ein ungültiger Index wird mit dem Rückgabewert -1 quittiert.

- `int bewerteVersuch(int versuch)` ermittelt die Anzahl s der schwarzen und die Anzahl w der weißen Stifte für den übergebenen Versuch und gibt diese als die Zahl $10s+w$ zurück. Der Aufruf `bewerteVersuch(491472)` führt bei dem Geheimcode 711574 also zu dem Ergebnis $2 \cdot 10 + 1 = 21$. Ein ungültiger Versuch (s.o.) wird mit dem Rückgabewert -1 zurückgewiesen.
- Eine Methode `boolean spielGewonnen()` prüft, ob der zuletzt abgegebene Versuch erfolgreich war (Rückgabewert `true`) oder nicht (Rückgabewert `false`). Falls es noch gar keine Versuche gab, weil das Spiel soeben erst gestartet wurde, soll ebenfalls `false` zurückgegeben werden.
- Mit der Methode `void spielZuruecksetzen()` lässt sich jederzeit wieder von vorn beginnen, d.h. alle Versuche werden gelöscht und es wird ein neuer Zufallscode erzeugt.

Alle `public`–Methoden (abgesehen von den Konstruktoren) müssen sich kreuz und quer durcheinander aufrufen lassen, ohne dass es Abstürze, Fehlfunktionen usw. geben darf. Weitere `private`–Methoden (auch bei den folgenden Aufgaben) dürfen Sie nach Belieben hinzufügen.

Für die ExpertInnen: Schaffen Sie es, die Bewertungsmethode mit einer linearen Laufzeitkomplexität zu implementieren? Bei einer allgemeinen Codelänge von n Ziffern sollte Ihre Routine also nur $O(n)$ viel Zeit benötigen.

Aufgabe 2: Erzeugen Sie eine Präsentations– bzw. Sicht–Klasse mit diesen `public`–Methoden:

- Der Konstruktor nimmt als Parameter eine zuvor erzeugte Modell–Klasse entgegen.
- Der Aufruf von `void denSpielerBegruessen()` führt zu einer freundlichen Willkommensmeldung, die dem Spieler zu Anfang präsentiert werden kann.
- Eine Methode `void druckeSpielverlauf()` erzeugt auf dem Bildschirm eine Darstellung aller bislang erfolgten Versuche (in der richtigen Reihenfolge) zusammen mit den zugehörigen Bewertungen, damit der Spieler aller Informationen für seinen nächsten Versuch zur Hand hat. Überlegen Sie sich auch eine passende Ausgabe für den Fall, dass noch gar keine Versuche vorliegen.
- `void spielerZumVersuchAuffordern()` zeigt auf dem Bildschirm einen Eingabehinweis an und informiert ihn zugleich darüber, um den wievielten Versuch es sich handelt.
- Die Methode `void demSpielerGratulieren()` dient zur Ausgabe einer abschließenden Meldung, wenn der Spieler zuvor bereits den Geheimcode geknackt hat.

Aufgabe 3: Erzeugen Sie eine Steuerungs–Klasse mit diesen `public`–Methoden:

- Der Konstruktor nimmt als Parameter eine zuvor erzeugte Modell– und eine ebenfalls zuvor erzeugte Sicht–Klasse entgegen.
- Ansonsten gibt es nur eine weitere öffentliche Methode, nämlich `void starteSpiel()`. Dort wird gesamte Spielverlauf abgewickelt. Nach der Rückkehr ist das Spiel beendet.
- In der statischen `main`–Methode erstellen Sie lediglich eine Modell–, Präsentations– und Steuerungs–Klasse und starten dann das Spiel.

Abgabe (d.h. Live–Demo inkl. Erläuterungen zum Programm) bis einschließlich zur letzten Vorlesungswoche Ende Januar 2017 während den Übungszeiten.